НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут прикладного системного аналізу

Кафедра системного проєктування

До захисту допущено:

Завідувач кафедри

_____ Вадим МУХІН

«___»_____20__ р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

спеціальності 122 «Комп'ютерні науки»

на тему: «Оптимізація симуляції міста в комп’ютерних іграх»

Виконав :
студент IV курсу, групи ДА-92
Титарчук Владислав Ігорович _____

Керівник:
асистент
Марков Дмитро Костянтинович _____

Консультант з нормконтролю:
доцент
Кирюша Богдан Анатолійович _____

Рецензент:
доцент
Аушева Наталія Миколаївна _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2023 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут прикладного системного аналізу

Кафедра системного проектування

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 "Комп'ютерні науки"

Освітньо-професійна програма – "Інтелектуальні сервіс-орієнтовані розподілені обчислювання"

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Вадим МУХІН

«___»_____20__ р.

ЗАВДАННЯ

на дипломну роботу студенту

Титарчука Владислава Ігоровича

1. Тема роботи «Оптимізація симуляції міста в комп’ютерних іграх», керівник роботи Марков Дмитро Костянтинович, асистент, затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом роботи – 10 червня 2023 р.

3. Вихідні дані до роботи

Дослідження та розвиток методів оптимізації симуляції міста в комп’ютерних іграх.

Програма для демонстрації методів оптимізація:

- Демонстрація методів оптимізації в комп’ютерних іграх.

4. Зміст роботи

1. Вступ
2. Аналіз проблеми та розгляд існуючих рішень
3. Постановка задачі та розгляд процесу симуляції
4. Створення гри та впровадження методів оптимізації у обраному середовищі розробки

5. Тестування та порівняння реалізованих методів оптимізації
6. Висновки

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

1. Презентація до захисту роботи.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Н. В., доцент		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Отримання завдання	08.12.2022	
2.	Дослідження поняття методів оптимізації у комп'ютерних іграх	13.04.2023	
3.	Ознайомлення з літературою і підготовка теоретичної частини роботи	27.04.2023	
4.	Огляд підходів до вирішення проблеми	06.05.2023	
5.	Розробка та тестування методів оптимізації ігор	10.05.2023	
6.	Виправлення помилок, знайдених при тестуванні	11.05.2023	
7.	Оформлення дипломної роботи	08.06.2023	

Студент Титарчук В.І.

Керівник Марков Д.К.

АНОТАЦІЯ

бакалаврської дипломної роботи Титарчука Владислава Ігоровича на тему: «Оптимізація симуляції міста в комп’ютерних іграх»

Комп’ютерні ігри є однією з сучасних галузей ПЗ, що активно розвивається і слугує не тільки у якості величезного за обсягом ринку розваг, але і одним із рушіїв розвитку комп’ютерної науки і техніки, як в сфері комп’ютерного обладнання, так і програмного забезпечення, зокрема, найрізноманітніших алгоритмів щодо генерації, моделювання та відтворення різних складних структур, які і розглядаються в цій роботі.

Метою даної дипломної роботи було на прикладі простої комп’ютерної гри-демонстрації продемонструвати методи такого моделювання та засоби їх оптимізації, зокрема створити систему моделювання міста з продуктивністю, задовільною на персональних комп’ютерах та наочно продемонструвати результати впроваджених методів оптимізації.

В ході даної роботи було досліджено вже існуючі приклади та зразки а також проведено ознайомлення з наукової літературою на тему та на основі цього сформовано список методів оптимізації, що підходять для впровадження і розроблено два прототипи гри – з імплементованимим методами та без них, а також проведено аналіз та продемонстровано відмінність у продуктивності між двома прототипами та зроблено висновки щодо ефективності та необхідності таких методів. У якості середовища розробки було використано ігровий рушій з відкритим вихідним кодом Godot Engine 4 та вбудовану мову програмування GDScript, засновану на Python.

Загальний обсяг роботи: 93 сторінок, 34 рисунки, 6 додатків, 19 джерел.

Ключові слова: Godot Engine, оптимізація, комп’ютерні ігри, симуляція, багатопотоковість, моделювання.

ABSTRACT

of a bachelor's thesis by Tytarchuk Vladyslav Ihorovych on the topic: "Optimization of city simulation in computer games"

Computer game development is one of the modern software industries that is actively developing and serves not only as a huge entertainment market but also as one of the drivers of computer science and technology development, both in the field of computer hardware and software, in particular, a variety of algorithms for generating, modeling and reproducing various complex structures, which are discussed in this paper.

The purpose of this thesis was to demonstrate the methods of such modeling and the means of their optimization on the example of a simple demonstration computer game, in particular, to create a city modeling system with satisfactory performance on personal computers and to demonstrate the results of the implemented optimization methods.

In the course of this work, existing examples and samples were studied, as well as the scientific literature on the topic was reviewed, and on this basis, a list of optimization methods suitable for implementation was formed and two game prototypes were developed - with and without implemented methods, as well as an analysis and demonstration of the difference in performance between the two prototypes and conclusions about the effectiveness and necessity of such methods were drawn. The open-source game engine Godot Engine 4 and the Python-based GDScript embedded programming language were used as the development environment.

Total volume of the work: 93 pages, 34 figures, 6 appendices, 19 sources.

Keywords: Godot Engine, optimization, computer games, simulation, multithreading, modeling.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ І СИМВОЛІВ

FPS – frames per seconds, кількість кадрів, що комп’ютер встигає обробити за секунду та відмалювати на моніторі, основний показник продуктивності в комп’ютерних іграх. Стандартом задовільної продуктивності є 60 кадрів на секунду.

CPU\ЦПУ – центральний процесор комп’ютера.

GPU\відеокарта – графічний процесор комп’ютера.

RAM\ОЗП – оперативна пам’ять комп’ютера.

Ігровий рушій – інтегрована середа розробки, що містить у собі усі необхідні інструменти для розробки комп’ютерних ігор та їх компіляції під кінцеву платформу.

Симуляція – (спроба) повторення в контрольованих умовах певної реальної речі, ситуації і процесу.

Комп’ютерна симуляція – те ж саме, але повторення відбувається у цифровому вигляді (програма).

Скрипт – невеликий фрагмент коду в окремому файлі, що має певні конкретні фунції.

Реплей – запис гри на ігровому рушії, що можна потім переглянути в «інтерактивному» форматі, проглядаючи інформацію про ігрові параметри в будь-який момент гри.

Спрайт – зображення в грі, що належить до якогось з ігрових об’єктів.

1. ВСТУП

Комп’ютерні ігри на сьогодні є не тільки однією з галузей індустрії розваг, що найбільш активно розвивається, але і передовим рушієм розвитку комп’ютерних технологій для пересічних користувачів загалом. З одного боку, щоб підтримувати інтерес користувачів, іграм необхідно постійно розвиватися та покращуватися, що веде до зростання реалістичності графічних ефектів у них, та, як наслідок – системних вимог, що вимагають все більш сучасних процесорів та відеокарт. То ж саме з прицілом на комп’ютерні ігри зазвичай розробляються та презентуються нові моделі подібної техніки від передових брендів користувацької електроніки. З іншого боку, комп’ютерні ігри дають розробникам необмежений простір для творчості та дозволяють пропрацьовувати глибину симуляції складних фізичних моделей та процесів, будь то пошкодження військової техніки від влучання снарядів, чи цілі світи, що активно живуть своїм життям, розвиваючись за певними ігровими законами та правилами. Такі ігри, фактично, мають цінність як з точки зору гравців, адже вирізняються своєю глибиною пропрацювання та реалізмом, так і з дослідницької точки зору, адже фактично являють собою візуалізовані математичні моделі певних складних явищ або процесів.

З іншого боку, гостро постає питання їх оптимізації, адже, не зважаючи на розмах та фантазію розробників, вони мають з пристойною продуктивністю запускатися на ПК пересічних користувачів, що вимушує розробників докладати значних зусиль, та шукати оригінальні методи оптимізації, які дозволять максимально якісно адаптувати саме їх гру під можливості ПК гравців, та при цьому не втратити ідеї, яка початково закладалася в гру.

Саме такому випадку і буде присвячена ця робота, в якій будуть розглядатися методи оптимізації такого складного процесу, як симуляції, з певним рівнем абстрактності, цілого міста, зокрема його жителів.

2. АНАЛІЗ ПРОБЛЕМИ ТА РОЗГЛЯД ІСНУЮЧИХ РІШЕНЬ

2.1. Історія проблеми

Симуляція міста, як і будь якої складної системи з реального життя, є і завжди було актуальним питанням в розробці комп'ютерних ігор, адже так чи інакше, ігри є відображенням нашого життя, основна задача яких – поставити гравця в нові для нього (такі, яких він не зустріне в реальному житті) умови, при цьому зробивши їх зрозумілими. На цьому тим чи іншим чином базуються усі комп'ютерні ігри. Гравцю має бути зрозуміла мотивація головного героя і його ворогів (якщо такі наявні), мета гри, базові правила та сутність ігрового світу в якому цей процес відбувається. І звісно ж, важко уявити собі світ, в якому були б повністю відсутні міста, населені жителями, або ж інші подібні системи (селища, космічні станції, тощо).

Відповідно до цього, логічним є припустити, що на даний момент, в розквіт індустрії комп'ютерних, а особливо мобільних ігор надзвичайно актуальною є проблема не лише симуляції таких систем, з високим рівнем достовірності, відповідно до запитів гравців, але і оптимізація таких систем до рівня, що дозволить їм з прийнятною продуктивністю працювати навіть на обмежених потужностях мобільних телефонів гравців, не викликаючи при цьому особливих нарікань на продуктивність.

Разом з тим, аналізуючи ринок комп'ютерних ігор, можна знайти доволі велику кількість зразків, що вже успішно виконують цю задачу, що наштовхує на висновок, про активне застосування в них певних методів, які і дозволяють досягти прийнятної продуктивності на ПК гравців, отже актуальним стає їх аналіз та виокремлення таких методів з метою їх подальшого застосування в роботі.

2.2. Розгляд ігор, в яких реалізована симуляції міста та пошук методів оптимізації в них

Питання симуляції міста, як основного оточення, у якому так чи інакше проходить ігровий процес, в ігровій індустрії є не новим і активно розвивається ще з кінця 80-х та початку 90-х років 20-го століття. Так, в 1989 році вийшла гра SimCity, в якій гравцеві доводилося будувати місто для проживання жителів, враховуючи при цьому їх потреби, особливості природнього ландшафту та різноманітні інші параметри, що, певною мірою цілком являло собою симуляцію цілого міста [1].

З того часу ця проблема активно досліджувалася та розвивалася як і в жанрі містобудівельних симуляторів, так і в багатьох інших, адже з розвитком ігрової індустрії та можливостей персональних комп'ютерів користувачів зростали і їх вимоги до ігор, зокрема, до реалістичності процесів та оточення в них, що виливалося у створення більших локацій-міст, наповнених величезною кількістю жителів та безліччю процесів, що протікають в них кожну секунду, при цьому за своїми системними вимогами залишаючись доступними для пересічних користувачів. На сьогодні, з огляду на тематику даної роботи, можна виділити три основні жанри з реальними прикладами ігор, та розглянути основні методи оптимізації, що застосовуються в них:

1. РПГ-ігри

В таких іграх місто, як правило, виступає в якості локації, де відбуваються події гри, а реалістичність його симуляції може сильно відрізнятися в конкретних іграх. Тим не менш, існують декілька конкретних прикладів, наприклад Cyberpunk 2077, або ж навіть серія ігор GTA, в яких симуляції міста приділяється особлива увага. Гравця, коли він знаходиться безпосередньо у місті, постійно оточує велика кількість жителів, кожен з яких прямує у своїх справах, дотримуючись при цьому правил дорожнього руху, як у пішому порядку, так і на автотранспорті.

Основним методом оптимізації тут є те, що місто симулюється не повністю, а лише в невеликій зоні навколо гравця, яку він безпосередньо спостерігає, при цьому за межами цієї зони симуляція повністю або частково відсутня і там навіть немає жителів. При цьому жителі не є персоналізованими, і кожного разу, додаючи нового жителя в зону симуляції, гра просто буде випадковим чином генерувати його параметри, а потім так само видаляти, не запам’ятовуючи. Завдяки цьому вдається досягти ефекту постійного життя міста, адже гравець завжди бачить лише «живу» його частину, і з його точки зору місто завжди буде повністю живим. [2]

Рисунок 1 – Скріншот гри GTA V з офіційного сайту розробника. [3]

2. Містобудівні симулятори

В таких іграх перед гравцем постає задача побудови та розвитку власного міста, основну увагу при цьому приділяють жителям та їх потребам. Сучасні представники жанру, наприклад гра Cities Skylines відстежує окремо кожного жителя від його народження до смерті, слідкуючи за його параметрами, положенням в ігровому світі, та іншими особливостями, якій гравець може переглянути в будь який момент часу. Звісно ж, така точна симуляція потребує великої кількості системних ресурсів, а оскільки якихось спеціалізованих

методів оптимізації в цій грі розробниками використано не було, то зі збільшенням свого міста та кількості жителів у ньому гравці прогнозовано зтикаються з проблемами з продуктивністю.

Основна методика оптимізації тут є суто графічною, тобто жителі не промальовуються на ігровій мапі, коли заходять, наприклад, в транспорт або будівлі, але все одно симулюються, що, в купі з відсутністю багатопотоковості, спричиняє велике навантаження на одне ядро процесору та призводить до зниження ФПС. [4]

Рисунок 2 – Скріншот гри Cities Skylines з цифрового майданчику Steam [5]

3. Реалістичні стратегії

Окремий жанр ігор хоч і не ставить перед нами задачу керуванням містом, але тим не менш, вкрай реалістично симулює усі процеси, що відбуваються у ньому, а також у навколишньому світі, що також ставить перед розробниками задачу пошуку шляхів оптимізації такого ресурсоємкого процесу. Чудовим прикладом цього може слугувати гра Dwarf Fortress, в якій на етапі створення ігрового світу генеруються та симулюються події, що відбувалися в ньому до моменту початку гри і які безпосередньо впливають на світ, а потім, вже під час ігрового процесу, після кожного ходу гравця, генеруються дії усіх істот у світі,

тобто відбувається постійна покрокова симуляція цілого світу, що певним чином є задачею, подібною до симуляції міста.

Тут також відсутні якісь загальні методи оптимізації, окрім як написання ефективного коду згідно з загальноприйнятих стандартів, патернів та практик, а також оптимізацій деяких деталей ігрових механік, що не видні гравцеві і випливають саме з особливостей конкретної гри, тобто не є придатними та повсюдного використання. Загалом же наслідком цього є те, що гра Dwarf Fortress, навіть маючи вкрай просту графіку у вигляді відображення ASCII-символів, кожен з яких позначає певну істоту чи об’єкт має доволі високі системні вимоги, зокрема процесор з частотою 2.4 ГГц у якості мінімальних вимог та процесор з частотою 4 ГГц у якості рекомендованих вимог [6].

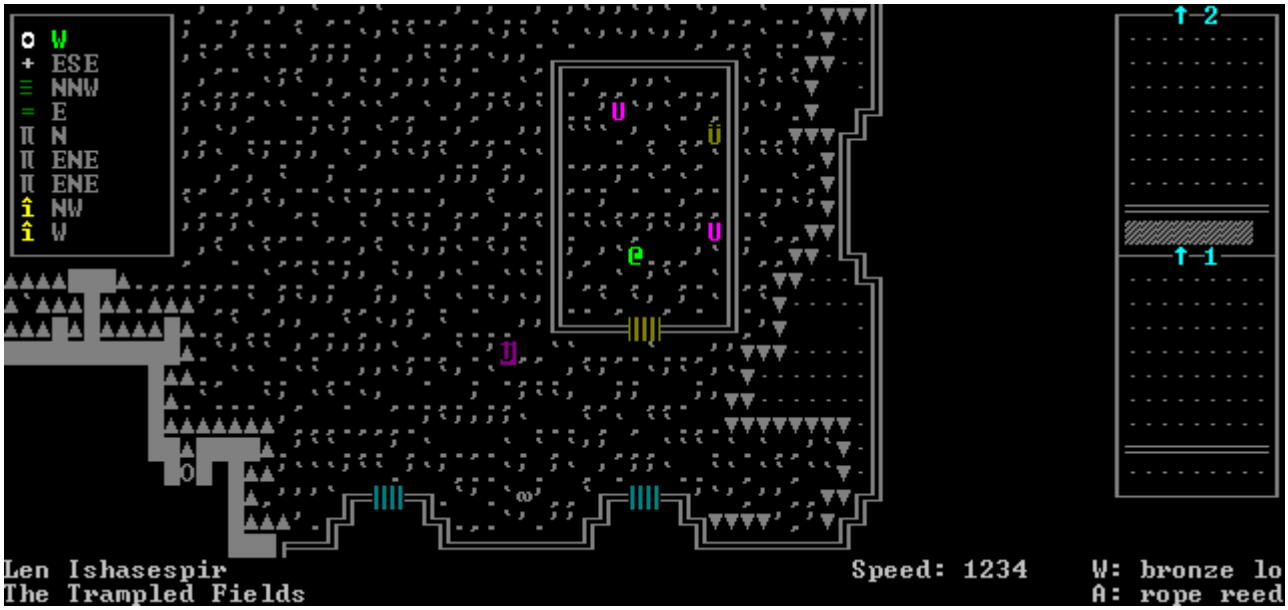

Рисунок 3 – Скріншот гри Dwarf Fortress з офіційного сайту розробника. [7]

2.3. Аналіз наявних методів оптимізації симуляції міста в комп’ютерних іграх

Аналізуючи на прикладі вищеназваних ігор методику симуляції міст або подібних до них систем у сучасних іграх, а також методи оптимізації подібних симуляцій, що використовуються в них, можна прийти до наступних висновків:

1. Повноцінних універсальних методів оптимізації для таких систем досі критично мало, і наслідком цього є те, що ігри з подібними симуляціями або мають незадовільну продуктивність, особливо при зростанні кількості симульованих об’єктів, або вдаються до хитрощів, щоб створити у гравця ілюзію симуляції, при цьому насправді не проводячи її. Це, в свою чергу, ще раз актуалізує питання проблеми, що розглядається в даній роботі та дає можливість в подальшому застосовувати на практиці розглянуті у ній методи.

2. Більшість розробників ігор приділяють увагу саме оптимізації конкретних ігрових механік, що випливає з особливостей тієї чи іншої гри, проте можна виділити й деякі загальні методи та підходи, а саме:

- Часткова симуляція: певні об’єкти та процеси, особливо ті, що не може спостерігати гравець, не симулюються в реальному часі, а обчислюються наперед за спрощеними формулами та просто зберігаються в такому стані, поки знову не почнуть симулюватися повністю.
- Проста графіка: небезпідставно вважаючи, що масштабна симуляція сама по собі робить гру достатньо цікавою, щоб захопити та втримати гравця в ній, розробники зазвичай вважають за краще зробити в грі стилізовану, але просту графіку, яка сама по собі не буде витрачати багато додаткових ресурсів ПК гравця.
- Випадкові величини та події: деякі аспекти симуляції також не симулюються повністю, а замість цього обраховуються випадковим чином на основі деяких параметрів, як, наприклад, шанс жителя міста захворіти, без реальних прорахунків контактів з вже інфікованими та інших параметрів, що мають значення в таких ситуаціях в реальному житті, але

при цьому створюючи ілюзію більш глибокої, пропрацьованої та реалістичної симуляції.

Це дає можливість говорити про актуальність проблеми оптимізації подібних симуляцій, а окрім того, дає певний базовий набір методик, що можуть застосовуватися в таких випадках. На додаток до цього, вважаю за доцільне також розглянути наступні методи, які, хоч і не знайшли поки що широкого застосування в індустрії комп'ютерних ігор, але при їх коректній реалізації цілком можуть бути вельми ефективними з точку зору оптимізації подібних симуляцій, а саме:

- Багатопотоковість: Більшість сучасних процесорів ПК мають декілька (від 2 до 32) ядер, кожен з яких розділяється на два потоки, при цьому розглянуті вище ігри зазвичай використовують лише одне з них для усіх необхідних обчислень. Якщо розподілити навантаження рівномірно між усіма ядрами, то це дасть значущий приріст в продуктивності. В основному такий метод є непопулярним через складність його реалізації та проблеми синхронізації потоків [8].
- Просунуте кешування: деякі обчислення можна спростити, не проводячи їх кожного разу в певних ситуаціях, а зберігаючи результати і повторно використовуючи їх, таким чином зменшуючи навантаження на обчислювальні потужності ПК [9].
- Оптимізований рендеринг: в рамках сучасних технологій комп'ютерних ігор, зокрема відкладеного рендерингу, можливо відмальовувати на екрані лише ті об'єкти, які безпосередньо потрапляють у кадр, таким чином зменшуючи навантаження на відеокарту і процесор [10].

Окрім цього, будуть враховані і локальні методи оптимізації, такі, як написання більш ефективного коду згідно стандартів обраної мови програмування та середовища розробки, оптимізація конкретних моментів гри з врахуванням реалізованих ігрових механік та інше. Саме ці методи будуть розглядатися у даній роботі, та, за можливості, якщо це дозволятиме обрана мова програмування та середовище розробки, реалізовані у грі-симуляції.

2.4. Шляхи розвитку або альтернативні методи оптимізації

Усі розглянуті вище методи оптимізації є, звісно ж дієвими та ефективними і їх впровадження є вкрай бажаним у будь-яких іграх, де присутня симуляція міста, але, тим не менш, вони всі є вже так чи інакше описаними і з точки зору дослідження технологій оптимізації не несуть нічого нового. Тому мною було вирішено кардинально доопрацювати один з цих методів, додавши у нього щось нове, що давало б змогу значно покращити оптимізацію в грі, при цьому не погіршуючи її реалістичність.

Як основу мною було вирішено взяти метод «Часткова симуляція» як такий, що має великий потенціал для розвитку, але разом з тим, і недоліки, які можна усунути. Основним його мінусом є те, що розробник наперед визначає області, в яких симуляція ведеться (саме місто, вулиця) і області, в яких симуляція не ведеться (всередині будівель), що так чи інакше створює у гравця відчуття фальшивості. Адже якщо в будинок не можна зазирнути, то, можливо, там нічого і не відбувається?

Для уникнення цієї проблеми мною пропонується новий метод, названий мною методом «Фрагментованої симуляції», сутність якого полягає в наступному:

1. В зоні, де в цей момент перебуває гравець (наприклад, вулиці міста) симуляція ведеться в повному об’ємі, з постійним оновленням стану усіх об’єктів згідно з заданих ігрових правил.
2. При переміщенні з однієї зони в іншу (захід в будівлю, тощо), жителі перестають симулюватися в основній зоні, але передають в цю нову зону певні дані про себе.
3. Коли гравець переходить в цю іншу зону, там, на основі записаних даних, пришвидшено прораховується симуляція до поточного моменту в грі, після чого починає йти в звичайному порядку. Симуляція ж в основній зоні в цей момент навпаки, припиняється, але дані про стан усіх об’єктів зберігаються.

Таким чином симуляція завжди ведеться одночасно лише в одній зоні – там, де знаходиться сам гравець, але при цьому він може в будь який момент перейти в іншу зону і там також почнеться симуляція, таким чином, буцімто вона велася тут увесь цей час завдяки пришвидшеному прорахунку.

Подібні технології в індустрії комп'ютерних ігор використовуються для системи записів та реплеїв, коли попередні матчі можна завантажити та переглянути, при цьому переходячи до буть якого часового проміжку та маючи доступ до інформації про об'єкти у грі. Але основна відмінність полягає в тому, що результат такого реплею відомий наперед, адже матч вже пройшов і просто є записаним, в той час як в грі симуляції від буде прораховуватися до моменту, коли гравець перейшов до цієї зони, а далі симулюватися вже в повному обсязі, з постійним оновленням результатів симуляції.

Переваги такої системи очевидні, адже вона дозволяє при збереженні ефекту оптимізації «Часткової симуляції» зробити симуляцію набагато більш реалістичної та детальною з точки зору гравця, не залишаючи місць, куди він не міг би зазирнути. Але вона має і певні недоліки, про наслідки та методи боротьби з якими також варто поговорити:

- Перевантаження ігрових об'єктів

Оскільки симуляція стає значно пропрацьованішою, то в ігрових об'єктів, наприклад, жителів, з'являється необхідність у більшій кількості ігрових механік, що веде до збільшення розмірів та перевантаження їх коду. Наприклад, в працівника мають бути наявні механіки і для переміщення по місту, і для роботи в офісі і для відпочинку у домі. Відповідно буде рости і кількість параметрів та змінних, що несе в собі кожен такий об'єкт, а це вже буде негативно вливати на продуктивність симуляції, адже такі об'єкти споживатимуть більше системних ресурсів.

Виходом з цієї ситуації, на мою думку, може бути методика розділення одного ігрового об'єкту симуляція на декілька, кожен з яких діє за своїми принципами та правилами та лише в своїй зоні, взаємозамінюючись при переході між зонами, тобто фрагментація самого об'єкта на декілька взаємопов'язаних.

Цей метод є доволі складним, а головне, потребує декількох типів зон в грі та великої кількості механік самого жителя, щоб мати ефект, адже інакше сама проблема теж буде не вкрай актуальною і не дасть значного негативного впливу на продуктивність, тому в рамках даної дипломної роботи реалізований не буде.
- Складність реалізації

Цей метод для свого функціонування потребує реалізації великої кількість ігрових механік та функцій що на перший погляд здаються зовсім не очевидними і тому його введення в гру може бути трудомістким та потребувати великої кількості часу, тому варто зважати на це, якщо реалізація проекту ведеться у стиснуті строки.

В цілому ж цей метод є прогресивним способом покращення продуктивності симуляції міста в комп’ютерній грі та дозволяє досягти значного приросту показників продуктивності без погіршення реалістичності та глибини опрацювання симуляції, що вигідно відрізняє його від методу «Часткова симуляція». Тому його реалізації та дослідженню, а також порівнянню з вже існуючими методами, в ході даної роботи буде приділена особлива увага.

2.5. Висновки до розділу

В цьому розділі було розглянуто питання проблеми оптимізації симуляції міста в сучасних комп’ютерних іграх та проаналізовано методи, що зазвичай застосовуються в таких випадках, або ж не застосовуються з певних причин, але все одно можуть бути ефективними.

Не зважаючи на те, що проблема оптимізації симуляції міста в комп’ютерних іграх є далеко не новою і існує вже принаймні кілька десятиліть, сам набір методів, що використовується в таких ситуаціях і досі залишається зовсім не вичерпним, і не вирішує в повній мірі задачу реалізації комплексної та реалістичної симуляції на потужностях персональних комп’ютерів пересічних користувачів, зазвичай обмежуючись лише створенням ілюзії такої симуляції для гравця, або ж і просто маючи нарікання на продуктивність з боку гравців.

Тим не менш, враховуючи методи оптимізації, що так чи інакше застосовуються у цих іграх, а також поєднуючи їх з іншими, не застосованими, але можливими до застосування, теоретично можливо досягти значного прогресу у вирішенні цієї проблеми та створити повноцінну, реалістичну і масштабну симуляцію, яка зможе з прийнятної продуктивністю працювати навіть на не найбільш потужних ПК.

Також було сформовано новий прогресивний метод оптимізації «Фрагментована симуляція», який хоч і має складності в реалізації, але технічно є більш досконалим, ніж його попередники який належить більш детально дослідити в ході виконання даної роботи

3. ПОСТАНОВКА ЗАДАЧІ ТА РОЗГЛЯД ПРОЦЕСУ СИМУЛЯЦІЇ

3.1. Умови реалізації та проведення внутрішньоігрової симуляції

Перш за все, необхідно чітко сформулювати критерії оцінювання продуктивності гри, себто ефективності впроваджених методів оптимізації при порівнянні продуктивності до та після їх впровадження, а також визначити саму сутність гри в контексті симуляції, тобто описати об’єкти та процеси які будуть реалізовані у грі, будуть симулюватися і, відповідно, підпадають під оптимізацію.

Очевидно, що симуляція у грі буде реалізовуватися на певному базовому рівні та зі значною долею абстрактності, адже неможливо в повній мірі точно просимулювати таку складну систему, як реальне місто та його жителів, але, тим не менш, симуляція включатиме в себе усі основні атрибути реальних міст (будівлі, жителі та їх переміщення, тощо) та загалом матиме доволі широкий перелік внутрішньоігрових механік, що дасть широкий простір для впровадження розглянутих раніше методів оптимізації.

3.2. Розгляд внутрішньоігрових об'єктів, механік та процесів симуляції

Місто в грі буде розглядатися та реалізовуватися як набір об’єктів, кожен з яких має певні механіки та функціонал для взаємодії з іншими об’єктами та ігровим оточенням. Об’єкти у грі доцільним буде поділити на три категорії, за їх властивостями та притаманними механіками:

- Абсолютні об’єкти. Їх параметри в продовж усієї симуляції залишаються незмінними, а їх властивості визначають сам порядок цієї симуляції та правила взаємодії інших об’єктів. Наприклад, ігрова карта та внутрішньоігровий час.
- Статичні об’єкти, положення яких є незмінним, і які мають певний набір параметрів, але не ініціюють взаємодії з іншими об’єктами і не змінюють свої параметри в продовж симуляції. Наприклад, житлові будинки та будівлі, де працюють жителі.
- Динамічні об'єкти, що мають логіку переміщення по карті, взаємодії з іншими об’єктами та активної зміни власних параметрів (положення на карті, статус, тощо). Основними представниками такого об’єкту є жителі.

Кожен з цих типів об’єктів в рамках даної симуляції буде мати декілька конкретних видів представників (наприклад, будівлі різного призначення, тощо), в рамках одного виду усі ігрові логіки та механіки є однаковими, але можлива відмінність у параметрах. Наприклад, житловий будинок може бути як на 100, так і на 200 жителів.

Враховуючи, що в грі має бути коректно та повноцінно реалізований бодай один базовий сценарій симуляції (житель, що вночі «спить» дома, а в день працює на роботі), видається доцільним, в першу чергу реалізація наступних ігрових об’єктів:

Таблиця 1 – Базові ігрові об’єкти за категоріями, їх механіки та параметри

Абсолютні об’єкти	**Параметри**	**Механіки**
Ігрова карта	- Мапа навігації для жителів (доступні та недоступні для проходження регіони)	- Генерація шляху між двома точками на мапі за запитом жителя
Внутрішньоігрова система часу	- Поточний час доби в грі	- Оповіщення внутрішньоігрових об’єктів про настання певного часу
Статичні об’єкти	**Параметри**	**Механіки**
Житловий будинок	- Положення входу на карті - Кількість жителів - Поточні жителі в будівлі	- Перенесення жителя з будівлі на карту - Перенесення жителя з карти в будівлю
Офісна робоча будівля	- Положення входу на карті - Кількість працівників - Поточні працівники в будівлі	- Перенесення жителя з будівлі на карту - Перенесення жителя з карти в будівлю
Динамічні об’єкти	**Параметри**	**Механіки**
Житель	- Ідентифікатор - Положення на карті - Поточна активність	- Активація певних дій з настанням деякого внутрішньоігрового часу - Пошук шляху до заданого об’єкту та проходження по ньому - Взаємодія зі статичними об’єктами та занесення себе в них

3.2. Моделювання та опис базового сценарію симуляції

Для кращого розуміння, розглянемо базовий сценарій симуляції та визначимо, як при ньому можуть взаємодіяти ігрові об’єкти:

1. О 8:00 за внутрішньоігровим часом житель виходить зі свого будинку та йде по місту до місця своєї роботи.
2. Дістався до місця роботи, він заходить до будівлі і працює там до 18:00, після чого направляється назад додому.
3. Дістався до своєї будівлі, він відпочиває до 8:00 за внутрішньоігровим часом, після чого увесь процес починається знову і так по колу.

Цей сценарій видається вкрай простим з логічної точки зору, але, тим не менш, навіть в ньому відбувається доволі багато подій на кожному з етапів, а саме:

1. Етап 1
    1. Внутрішньоігрова система часу оповіщає об’єкти про те, що настала 8:00.
    2. Будинок проживання жителя «створює» його на ігровій карті, на основі записаних раніше параметрів з масиву усіх жителів, що живуть у цьому будинку.
    3. Житель «звертається» до ігрової карти та на основі даних з неї прокладає собі маршрут до місця роботи (інші динамічні об’єкти в даному випадку не враховуються).
    4. Житель проходить за маршрутом, та, дістався до місця роботи, заходить в нього, «зникаючи» з карти і записуючи свої дані у масив працівників місця роботи.
2. Етап 2
    1. Житель перебуває на місці роботи, його робота там певним чином симулюється. Поки гравець не переходить всередину цієї будівлі, симуляція ведеться за спрощеними алгоритмами, але як тільки він заходить

– детально прораховується до поточного моменту у грі та продовжується далі вже в онлайн-режимі.

2. Внутрішньоігрова система часу оповіщає об’єкти про те, що настала 18:00.

3. Місце роботи жителя «створює» його на ігровій карті, на основі записаних раніше параметрів з масиву усіх жителів, що тут працюють.

4. Житель «звертається» до ігрової карти та на основі даних з неї прокладає собі маршрут до дому (інші динамічні об’єкти в даному випадку не враховуються).

5. Житель проходить за маршрутом, та, дістався до місця проживання, заходить в нього, «зникаючи» з карти і записуючи свої дані у масив жителів цього будинку.

3. Етап 3

1. Перебування жителя в будинку певним чином симулюється (в найбільш базовій реалізації – ніяк, житель просто чекає початку робочого дня).

Як видно з описаної вище схеми, навіть такий базовий сценарій симуляції має велику кількість кроків, механік та взаємодій, кожна з яких може бути певним чином реалізована та оптимізована як з точки зору безпосередньо коду, так і використання інструментів обраного середовища розробки (ігрового рушія).

3.3. Постановка задачі щодо реалізації та оптимізації симуляції

Таким чином, загальна задача щодо програмної реалізації симуляції та її оптимізації буде такою:

1. Реалізація описаних ігрових об’єктів у вибраному середовищі розробки.
2. Поєднання їх і загальну систему, налаштування симуляції.
3. Тестування симуляції, пошук логічних помилок та недоліків, їх виправлення.
4. Тестування продуктивності симуляції.
5. Аналіз можливості впровадження визначених раніше методів оптимізації.
6. Впровадження визначених можливими до впровадження оптимізації.
7. Тестування оптимізованої симуляції та порівняння показників продуктивності з попередніми.

Після цього буде можливо зробити висновки про ефективність проваджених методів та доцільність їх впровадження, що і є основною метою даної роботи.

З теоретичної точки зору, можна виділити три основні задачі:

1. До початку програмної реалізації – більш детально розглянути та проаналізувати наведені вище методи оптимізації, які плануються до реалізації.
2. В продовж програмної реалізації – збір статистики продуктивності, та пояснення реалізації ігрових об’єктів, механік, та реалізацій у вибраному середовищі розробки
3. Після програмної реалізації – аналіз отриманих даних, та формування висновків щодо ефективності впроваджених методів.

3.4. Параметри та методика оцінки продуктивності симуляції

Оцінка продуктивності гри до та після впровадження методів оптимізації буде проводитися за стандартної для комп'ютерних ігор процедурою, а саме:

1. Буде розроблено дві версії гри, повністю однакові за своєї логікою симуляції та графічною частиною, але з застосуванням та без застосування визначених методів оптимізації.
2. Кожна з версій буде скомпільована під цільову платформу (в даному випадку – ОС Windows) за допомогою обраного середовища розробки однакової версії.
3. Обидві версії будуть тестуватися на одному і тому ж ПК, за однакових умов (без інших запущених програм, окрім вбудованого Диспетчера задач).
4. Параметри, за якими буде відбуватися оцінка, будуть наступними:

- ФПС в грі, що визначатиметься за допомогою допоміжного коду та об'єктів в самій грі (інструменти ігрового рушія), у одиницях.
- Споживання процесом гри ресурсів ЦП з Диспетчеру задач, у відсотках на кожне ядро.
- Споживання процесом гри ресурсів ОЗП з Диспетчеру задач, у мегабайтах.
- Споживання процесом гри ресурсів відеокарти з Диспетчеру задач, у відсотках.

5. Тестування проходитиме впродовж усього базового сценарію симуляції (див. пункт 2.2), з фіксуванням результатів в різних моментах сценарію:

1. Початок, до виходу жителів з будинків.
2. «Годину пік», коли більшість жителів вийшла зі своїх будинків та рухається вулицями міста.
3. Середину робочого дня.
4. Вечірню годину пік, коли жителі залишають робочі будівлі.

Після цього отримані результати будуть проаналізовані та на їх основі – зроблено висновок про ефективність впроваджених методів оптимізації.

3.5. Розгляд ігрового рушія Godot Engine в контексті реалізації гри-симуляції

Отож, перш за все необхідно коротко ознайомитися з обраним ігровим рушієм, принципами та системами його роботи, інструментами, які він надає та дослідити можливі методи реалізації на ньому визначених раніше ігрових об’єктів, параметрів, та механік. При цьому, оскільки метою роботи є дослідити саме універсальні методи оптимізації, не прив’язані до конкретного рушія чи мови програмування, то вони будуть розглянуті далі, але саме у контексті їх реалізації для обраних механік гри, а не з точки зору можливостей рушія.

Godot Engine є безкоштовним ігровим рушієм з відкритим вихідним кодом, що активно розвивається спільнотою розробників ентузіастів на пожертви своїх фанатів. Він має на меті бути легким в освоєнні та використанні, але при цьому універсальним ігровим рушієм, що містить у собі усі необхідні для розробки інструменти ігор будь-яких жанрів та їх компіляції під різні платформи [11].

Однією з найбільш вражаючих властивостей рушія можна назвати його компактність - весь рушій, у версії для ОС Windows являє собою усього один виконуваний EXE-файл вагою менше ніж 100 мегабайтів. Серед інших переваг можна назвати простоту, але разом з тим потужність інструментарію рушія, включно з вбудованим редактором коду, який підтримує власну мову програмування GDScript, засновану на Python, а також можливість встановлення модулів на використання при розробці інших мов (C# та C++/C, зокрема), вбудовані інструменти для роботи з анімаціями, ігровими плитковими картами та інший функціонал [12].

Недоліками ж в першу чергу є проблемна в деяких сценаріях продуктивність, особливо в іграх з трьохвимірною графікою, а також через застосування порівняно повільної мови програмування GDScript для розробки, та обмежений інструментарій, який, хоч і зростає з кожною новою версією рушія, все ще не має деяких корисних функцій.

Для розробки програмної частини було обрано останню стабільну на момент роботи над проектом версію рушія 4.0.0. Це дає можливість отримати доступ до багатьох нових інструментів та функцій, хоч і зменшує кількість доступної навчальної інформації, якої набагато більше по попередній, але все ще актуальній версії 3.5.2. У якості мови програмування було обрано безпосередньо вбудовану мову GDScript, як найбільш краще інтегровану з іншими компонентами та функціями рушія. Ця мова має меншу продуктивність в складних сценаріях, ніж C# або C++, оскільки заснована на мові програмування Python, але в даному випадку це не має критичного значення, оскільки метою роботи є показати саме порівняльний ефект від впровадження методів оптимізації, а не максимальні абсолютні результати продуктивності, обрані методи ж є актуальними, дієвими та можливими для впровадження безвідносно до обраного ігрового рушія або мови програмування загалом.

Уся робота з рушієм проходить у редакторі, що містить усі необхідні інструменти для роботи над грою – редактор ігрових об’єктів та їх ієрархії на сцені, редактор властивостей самих об’єктів, файловий менеджер ігрових ресурсів, вбудований редактор коду та багато інших інструментів. Деякі елементи інтерфейсу є не зовсім зручними (так, редактор коду відкривається не на весь екран, а замість області перегляду сцени, при цьому частину його займає сам список наявних у проекті файлів коду, що сильно зменшує загальну область відображення самого коду), але загалом є доволі інтуїтивно зрозумілим та простим у освоєнні.

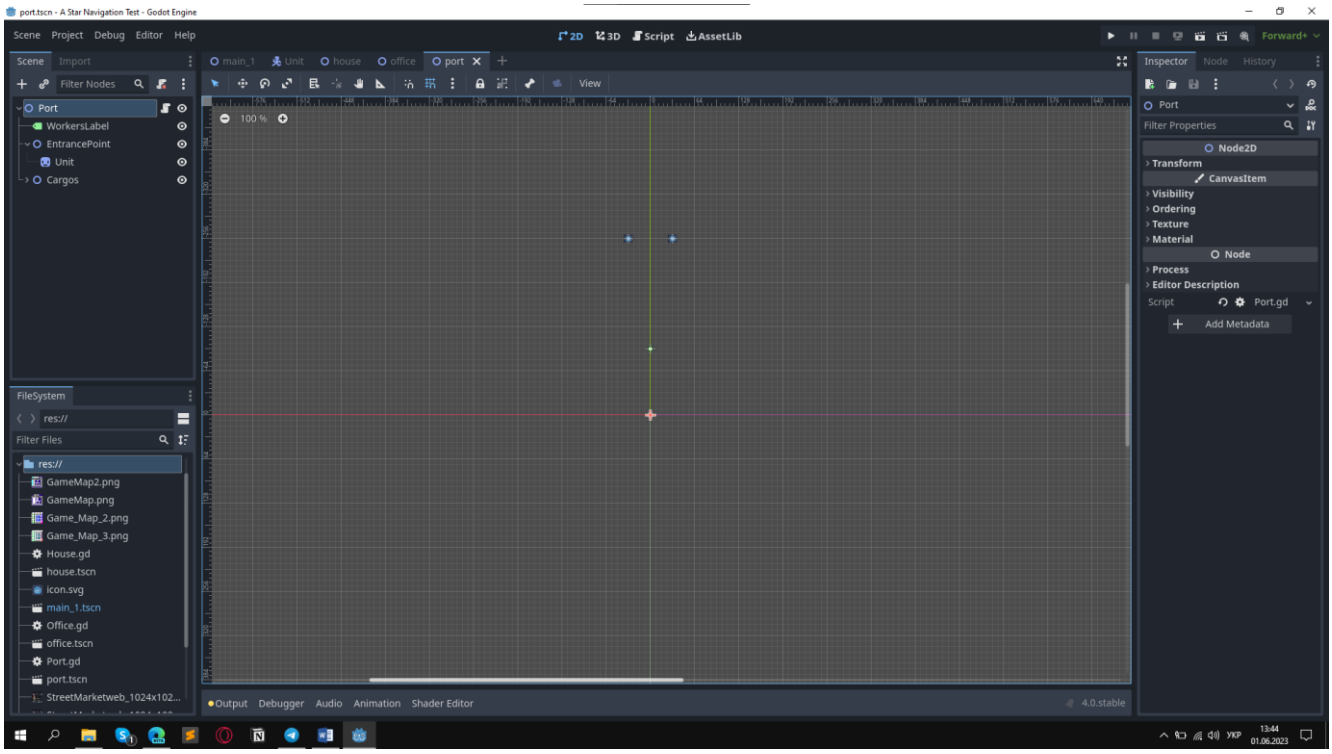

Рисунок 4 – Ігровий рушій Godot Engine

Розглянемо тепер безпосередньо ті інструменти та властивості рушія, які дозволяють реалізувати обрані для тестування базового сценарію об’єкти та механіки симуляції.

Основною ігровою механікою рушія Godot Engine, як і багатьох інших є «Сцена» – простір, всередині якого об’єкти можуть взаємодіяти між собою, коли вона активна. Наприклад, один рівень або локація в грі, головне меню, тощо [13].

Всередині сцени розміщуються з самого початку, або створюються в певні моменти через код будь-які необхідні ігрові об’єкти – оточення, головний герой, предмети, та усе інше, що так чи інакше присутнє в грі. При цьому, звісно ж, кількість сцен є необмеженою, а в відповідні ігрові моменти гравець може бути переміщений між сценами (вийти в меню, перейти на наступний рівень, тощо).

Об’єкти в грі представляють собою набори «вузлів», пов’язаних у єдину структуру, подібну до дерева, де коренем також виступає певний вузол, як правило, відповідальний за логіку і структуру самого об’єкту. «Вузли» фактично являють собою базові функціональні одиниці, і кожен тип має своє конкретне призначення. Так, наприклад, вузол «Sprite» відповідає за відображення якогось зображення, а вузол «Timer» може вести відлік часу та оповіщати інші вузли і

об'єкти, коли пройшов певний проміжок. При цьому властивості вузлів в об’єктах можна гнучко редагувати, в тому числі їх положення відносно один одного та загалом у просторі, та інші, більш специфічні параметри [13].

Особливістю ж рушія Godot Engine є те, що сцени також фактично являють собою окремі об’єкти або вузли, просто збережені розробником як окрема сцена, і доступні для подальшої взаємодії. Це зручно ти, що дозволяє створювати, наприклад, багато копій однієї і тієї ж сцени, кожна з яких буде мати свої параметри, але при цьому однакові базові механіки. Наприклад, жителі в цій роботі. При цьому достатньо один раз змінити базову сцену, щоб так само змінилися усі її копії у грі, а також не задумуватися про внутрішню структуру об’єктів у сцені гри – там води будуть представлені як суцільні вузли-сцени, без відображення вузлів всередині них. Таким чином гра загалом являє собою набір сцен-вузлів, кожна з яких є деревовидною структурою з інших об’єктів-сцен-вузлів, які динамічно змінюються за заданими розробником механіками, створюючи таким чином ігровий процес. Цей принцип, незвичний для інших рушіїв, де поняття сцени, ігрового об’єкту та його базових компонентів (вузлів) зазвичай чітко розділені, має як переваги (уніфікація ігрових об’єктів та систем, єдині та зрозумілі для розробника принципи їх роботи, можливості гнучкого налаштування та зручного редагування об’єктів), так і недоліки (заплутаність на перших порах, відсутність чіткого диференціювання, неоднозначність у можливих реалізаціях тих чи інших ігрових механік і т.п.), але загалом є доволі зручним і швидким в освоєнні та потужним за своїм функціоналом у подальшому використання.

Іншим важливим інструментом рушія є, звісно ж, кодова частина. В редакторі сцени розробник має можливість до кожного об'єкту прикріпити скрипт, який буде визначати його поведінку, окрім тих механік, що походять з самого типу вузлу.

3.6. Розгляд реалізації обраних методів оптимізації в грі-симуляції

Розглянемо тепер більш детально методи оптимізації, що плануються до реалізації, особливо в контексті запланованих механік та об’єктів, та з точки зору можливості їх реалізації в обраному середовищі розробки:

1. Часткова симуляція: певні об’єкти та процеси, особливо ті, що не може спостерігати гравець, не симулюються в реальному часі, а обчислюються наперед за спрощеними формулами та просто зберігаються в такому стані, поки знову не почнуть симулюватися повністю.

Цей метод, завдяки своїй простоті реалізації, ефективності та універсальності, на мою думку, є однім з основних методів, що мають бути впровадженими у першу чергу. В контексті гри-демонстрації, найбільш практичної для реалізації виглядає схема, коли подібним чином відбувається уся симуляція усередині статичних об’єктів, а саме:

1. Динамічний об’єкт (житель) підходить до статичного об’єкту (будівлі) та взаємодіє з ним.
2. Житель зникає з карти міста, як логічний і як ігровий об’єкт.
3. Замість симуляції в середині будівлі, параметри жителя відразу оновлюються на певну кількість годин уперед (наприклад, 8 годин на роботі), та записуються у масив цієї будівлі.
4. Через певний час створюється житель з такими параметрами на ігровій карті, і видаляється з масиву будівлі.

Це дозволить значним чином оптимізувати симуляцію, фактично не симулюючи більшість жителів велику частину часу, але роблячи це непомітно для гравця.

2. Фрагментована симуляція: об’єкти симулюються лише в тій зоні, де зараз знаходиться гравець, але при цьому є можливість переходити в інші зони, де при цьому відбувається швидкий прорахунок стану усіх об’єктів до поточного моменту, а далі йде їх звичайна симуляція, що робить гру більш реалістичною.

В контексті гри-демонстрації, найбільш практичної для реалізації виглядає схема, коли подібним чином відбувається уся симуляція усередині офісних будівель, а саме:

1. Динамічний об’єкт (житель) підходить до статичного об’єкту (будівлі) та взаємодіє з ним.
2. Житель зникає з карти міста, як логічний і як ігровий об’єкт.
3. Симуляція всередині будівлі не відбувається, а просто прораховується наперед.
4. Якщо в будівлю заходить гравець, то симуляція прораховується до поточного моменту після чого починає вестися у звичайному режимі
5. По завершенню робочого дня дані усіх працівників оновлюються згідно з результатами симуляції і вони полишають будівлю.

3. Проста графіка: небезпідставно вважаючи, що масштабна симуляція сама по собі робить гру достатньо цікавою, щоб захопити та втримати гравця в ній, розробники зазвичай вважають за краще зробити в грі стилізовану, але просту графіку, яка сама по собі не буде витрачати багато додаткових ресурсів ПК гравця.

Незважаючи на те, що термін «проста графіка» можна трактувати дуже широко – від текстового виводу у консоль, символи якого формують певне зображення, до 3Д графіки, яка не є самою сучасною, в даному випадку було вирішено обрати проміжний варіант – 2Д графіку, де кожен об’єкт являє собою окремий спрайт, як то житель, будівля чи сама ігрова мапа, без освітлення, тіней та інших графічних прикрас. Такий варіант дозволить, з однієї сторони, зробити графіку доволі приємною для гравця, а з іншої, «розвантажити» ПК гравця, вивільнивши таким чином системні ресурси на саму симуляцію. Чудовим наочним прикладом може слугувати гра Mini Motorways, що має схожий жанр і приємний візуальний стиль.

Рисунок 5 – Скріншот з гри Mini Motorways

4. Випадкові величини та події: деякі аспекти симуляції також не симулюються повністю, а замість цього обраховуються випадковим чином на основі деяких параметрів.

Так, наприклад, якщо в грі буде наявна механіка захворювань жителів, то шанс жителя міста захворіти може визначатися кожен день випадковим чином, просто з певною ймовірністю, без реальних прорахунків контактів з вже інфікованими та інших параметрів, що мають значення в таких ситуаціях в реальному житті, таким чином не сильно навантажуючи ПК, але при цьому створюючи ілюзію більш глибокої, пропрацьованої та реалістичної симуляції. Втім, це потребує реалізації в неоптимізованій версії гри цих доволі складних механік для їх подальшого порівняння.

5. Багатопотоковість: Більшість сучасних процесорів ПК мають декілька(від 2 до 32) ядер, кожен з яких розділяється на два потоки, при цьому розглянуті вище ігри зазвичай використовують лише одне з них для усіх необхідних обчислень. Якщо розподілити навантаження рівномірно між усіма ядрами, то це дасть значущий приріст в продуктивності. В основному такий метод є непопулярним через складність його реалізації та проблеми синхронізації потоків.

Основними об’єктами, що створюють навантаження на процесор є саме жителі, через їх велику кількість та динамічну природу (постійне обчислення положення на карті, та інших параметрів). Таким чином, навіть якщо можна розпаралелити лише 50% обчислень, що здійснюються цими жителями, використавши середні для персональних ПК 4 потоки, то за законом Амдала отримаємо наступний приріст продуктивності:

$$S = 1/(0.5 + (0.5/4)) = 1.6 \text{ разів.}$$

Тобто буде можливо приблизно ж з тією продуктивністю симулювати в 1.5 рази більше жителів, та додаткові статичні об’єкти для них (кількість об’єктів порівняно з кількістю жителів можна вважати незначною, приблизно 1 на 1000), або ж значно збільшити продуктивність симуляції без зменшення кількість об’єктів у ній. Необхідно дослідити рушій на можливість реалізації такої механіки та за можливості реалізувати її.

6. Просунуте кешування: деякі обчислення можна спростити, не проводячи їх кожного разу в певних ситуаціях, а зберігаючи результати і повторно використовуючи їх, таким чином зменшуючи навантаження на обчислювальні потужності ПК.

Припустимо, що певна кількість жителів працює на певній роботі, де тривалість робочого дня та зарплатня на годину є сталими. Замість того, щоб кожен день для кожного жителя вираховувати його зарплатню, шляхом перемноження зарплатні на годину на тривалість робочого дня, можна обчислити цю величину один раз і зберегти в параметрах місця роботи, і просто «видавати» цю зарплатню працівникам, без її постійного обчислення.

7. Оптимізований рендеринг: в рамках сучасних технологій комп’ютерних ігор, зокрема відкладеного рендерингу, можливо відмальовувати на екрані лише ті об’єкти, які безпосередньо потрапляють у кадр, таким чином зменшуючи навантаження на відеокарту і процесор.

З логічної точки зору ця техніка є доволі простою – маючи положення об’єкта в грі та його розміри, а також положення і зону відображення камери (того, що бачить гравець на моніторі), можемо легко порахувати чи видно об’єкт

у кадрі і, відповідно, рендерити його або ні. Але з технічного боку тут виникає багато питань, адже необхідно кожен кадр робити перевірку для потрапляння кожного об’єкту в грі у кадр, що саме по собі є великими витратами обчислювального ресурсу, аж особливо у випадку, коли це реалізується не в коді самого рушія, а скриптовою мовою в середовищі розробки. Як і у випадку з багатопотоковістю, спочатку дослідити рушій на можливість реалізації такої механіки, а вже потім, за можливості, реалізувати її.

Таким чином, з урахуванням даного аналізу інструментарію для розробки та самих методів оптимізації, можна зробити попередні висновки та план щодо їх реалізації у грі:

- Метод «Проста графіка» буде реалізовано за замовчуванням, щоб не витрачати час розробки на реалізацію двох різних графічних стилей.
- Методи «Часткова симуляція», та «Просунуте кешування», вірогідніше за все можуть бути без проблем реалізовані в рамках роботи з даним рушієм, чому і буде приділена основна увага.
- Метод «Фрагментована симуляція буде реалізований на заміну методу «Часткова симуляція» в окремій версії гри для подальшого порівняння їх продукивності.
- Метод «Випадкові величини та події» потребує для свого розгляду реалізації в неоптимізованій версії гри доволі складних механік та процесів для їх подальшого порівняння з оптимізованою версією, що, через обмежений час на розробку навряд чи може бути реалізовано, тож цей метод, вірогідніше за все, буде розглядатися лише з теоретичної точки зору.
- Методи «Багатопотоковість» та «Оптимізований рендеринг» потребують дослідження на предмет того, чи є вони вже впровадженими у самому рушії і якщо так, то ці пункти роботи можна вважати успішно виконаними, в противному ж випадку метод «Багатопотоковість» є сенс спробувати реалізувати у самій грі, метод «Оптимізований рендеринг» же через свої технічні особливості навряд-чи доцільно реалізовувати таким чином.

Таким чином, визначивши технічні особливості рушія, на якому передбачається реалізація проекту, та методів оптимізації, що плануються для впровадження, та їх поєднання між собою, можна перейти безпосередньо до технічної реалізації проекту.

3.7. Висновки до розділу

В цьому розділі було проведено коротке ознайомлення з ігровим рушієм Godot Engine на якому планується подальша програмна реалізація проекту, а також визначено конкретні приклади того, як саме обрані методи оптимізації можуть буду впроваджені в гру-симуляцію та продемонстровані на базовому сценарії.

З розглянутої інформації можна зробити висновок про те, що ігровий рушій Godot Engine є цілком функціональним середовищем розробки, що дозволяє реалізувати усі задумані для базового сценарію ігрові об’єкти та сценарії їх взаємодії, і при цьому значно спрощуючи розробку у порівнянні з реалізацією гри з нуля на певній мові програмування.

Також, навіть в базовому сценарії знайшлося місце для реалізації усіх визначених методів симуляції, що говорить про те, що даних сценарій виконує умови достатності з точки зору можливості демонстрації на ньому методів оптимізації, і при цьому необхідності з точки зору показу хоча б абстрактної і наближеної, але все ж симуляції міста, таким чином являючись оптимальним у рамках даної роботи. При цьому варто зауважити, що в той час, як більшість з них необхідно і доцільно впроваджувати безпосередньо в самій грі, частина методів скоріше потребують впровадження напряму в ігровий рушій, а інакше ефективність їх реалізації буде сумнівною, тому окрім впровадження необхідно буде також провести і їх дослідження вже безпосередньо у готовій грі.

В цілому ж проведений аналіз дозволяє зробити висновок про можливість та потенційну успішність реалізації проекту з урахуванням умов взаємодії усіх його компонентів, як то ігровий рушій, сама гра та методи оптимізації в ній, і дає можливість перейти безпосередньо до програмної частини роботи.

4. СТВОРЕННЯ ГРИ ТА ВПРОВАДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ В ОБРАНОМУ СЕРЕДОВИЩІ РОЗРОБКИ

В даному випадку процес створення гри доцільно почати з процесу створення усіх описаних раніше об’єктів симуляції, задання їм необхідних параметрів та іншого, а вже потім почати їх з’єднання у єдину систему, яка і буде являти собою готову гру. Втім, реалізація їх буде відбуватися у тому порядку, який дозволяє додавати до системи нові об’єкти послідовно, один за одним, при цьому нарощуючи функціонал системи і не маючи необхідності внесення значних змін до реалізованих вже раніше систем. З урахуванням списку об’єктів, необхідних до реалізації, оптимальний порядок видається наступним:

1. Житель.
2. Ігрова карта.
3. Внутрішньоігрова система часу.
4. Житловий будинок.
5. Офісна будівля.

Після цього буде можливо успішно об’єднати усі об’єкти в єдину систему гри-симуляції, готову до тестування та впровадження подальших методів оптимізації. Відповідно до плану роботи, буде створено дві версії гри для подальшого тестування, з впровадженими методами оптимізації та без них, для подальшого порівняння та тестування продуктивності. Порядок впровадження методів оптимізації буде наступним:

1. Проста графіка (ще на етапі розробки)
2. Багатопотоковість (якщо можливо)
3. Оптимізований рендеринг (якщо можливо)
4. Випадкові величини та події (не реалізується програмно)
5. Просунуте кешування.
6. Часткова симуляція
7. Фрагментована симуляція (окрема версія замість Часткової симуляції)

З урахуванням цього плану можна починати безпосереднє виконання робіт.

4.1. Створення об’єктів симуляції

1. Житель

Як вже було сказано в описі ігрового рушія Godot, усі об’єкти в ньому являють собою ігрові вузли (ноди) різних типів, пов’язані собою у структуру типу дерева. Певні об’єкти для зручності можна виокремити у сцени, таким чином в інших сценах вони будуть відображатися як окремі вузли, без урахування їх внутрішньої структури, що є доволі зручним та дозволяє не перевантажувати структуру ігрової сцени. За цим принципом будуть реалізовані усі об’єкти в грі.

У якості основи для реалізації жителя було вирішено обрати вузол типу Node2D – базовий вузол у двовимірному просторі, який має координату та інші параметри, що відносяться до його положення та орієнтації у просторі, а також методи, пов’язані з цим. В даному випадку він буде являти собою логічний контейнер для збирання усіх інших необхідних вузлів в одне ціле, а також використовуватися як точка відліку для визначення координат всередині сцени жителя та положення жителя на ігровій карті. До нього ж прикріплено і скрипт з реалізацією функціонала та механік даного типу ігрових об’єктів [14].

Також було застосовано інші типи вузлів, необхідні для реалізації цього об'єкта, а саме:

Sprite2D – базовий вузол для відображення зображень, використовується для відображення жителів на ігровій карті відповідним зображенням [15].

NavigationAgent2D – вузол, що відповідає за навігацію, дозволяючи налаштовувати параметри навігації для цього типу об’єктів, будувати маршрут та рухатися по ньому [16].

Timer – вузол для відліку часу за таймером та оповіщення про це інших вузлів об’єкту, необхідний для коректної роботи часових затримок в деяких процесах [17].

Label – вузол для виведення тексту на екран, в даному випадку – для відображення поточної кількості грошей у жителя [18].

Таким чином, структура окремої сцени жителя в ігровому редакторі буде виглядати наступним чином (при цьому в сцені самої гри вона буде представляти собою просто один вузол типу «Житель»):

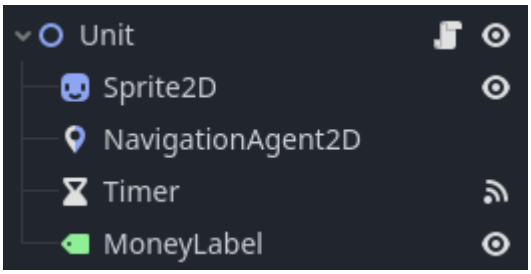

Рисунок 6 – Структура ігрового об’єкту жителя зсередини.

Окрім того, звісно ж, основним компонентом цього ігрового об’єкта, як і будь якого іншого, є скрипт, що відповідає за його поведінку в грі та взаємодію з іншими ігровими об’єктами. Розглянемо тут лише основні моменти коду, які дають загальне уявлення про роботу об’єкту.

Перш за все, ми створюємо усі змінні та параметри, необхідні для роботи об’єкта, а також підключаємося до інших вузлів та об’єктів, як то система навігації та власний навігаційний агент, а також система часу:

```
extends Node2D

@onready var time_system = $"../../Camera/TimeLabel"
@onready var nav_reg =  $"../../NavigationRegion2D"
@onready var nav = get_node("NavigationAgent2D")
@onready var timer = get_node("Timer")
@onready var money_label = get_node("MoneyLabel")

var speed = 250
var path = []
var map
var pathing_target = Vector2(0, 0)
var money = 0

var finished = false
var status = "staying_home"

var home : Node2D
var workplace : Node2D
var destination : Node2D
```

Рисунок 7 – Створення параметрів ігрового об’єкта «Житель»

Далі основними функціями для роботи об’єкту є побудова ними шляху для переміщення та власне переміщення цим шляхом:

```
func _update_navigation_path(start_position, end_position):
    path = NavigationServer2D.map_get_path(map, start_position, end_position, true)
    path.remove_at(0)
    set_process(true)
func move_along_path(distance):
    var last_point = self.position
    while path.size():
        var distance_between_points = last_point.distance_to(path[0])
        if distance <= distance_between_points:
            self.position = last_point.lerp(path[0], distance / distance_between_points)
            return
        distance -= distance_between_points
        last_point = path[0]
        path.remove_at(0)
    self.position = last_point
    set_process(false)
```

Рисунок 8 – Функції навігації ігрового об’єкта «Житель»

Оскільки у цій версії гри житель симулюється повністю, то він сам має мати можливість «вирішувати» коли йому вирушати на роботу або повертатися додому. Для цього реалізуємо функцію для взаємодії з ігровою системою часу та функцію для вибору жителем нової цілі і початку руху з певною затримкою:

```
func _on_time_label_time_signal(hours):
    if hours % 2 == 0:
        randomize()
        timer.wait_time = randf_range(1, 10)
        timer.start()
    if hours % 2 == 1:
        randomize()
        timer.wait_time = randf_range(1, 10)
        timer.start()
```

Рисунок 9 – Функція для взаємодії жителя з ігровою системою часу

```
func _on_timer_timeout():
    if(status == "staying_home"):
        destination = workplace
        home.living -= 1
        status = "going_to_work"
        finished = true
        start_path()
    if(status == "working"):
        destination = home
        workplace.workers -= 1
        status = "going_to_home"
        finished = true
        start_path()
```

Рисунок 10 – Функція для вибору жителем нової цілі та початку руху до неї

Реалізуємо і функціонал для симуляції жителя в середині офісної будівлі. Він буде нескладним і полягатиме просто у періодичному руху по офісу з постійним начисленням зарплатні.

```
func on_path_finished():
    await get_tree().create_timer(randi_range(5, 10)).timeout
    var working_destination = Vector2(randi_range(0, 512), randi_range(0, 512))
    destination = working_destination
    start_path()

func get_salary():
    while true:
        await get_tree().create_timer(1).timeout
        money += workplace.hour_salary/60
```

Рисунок 11 – Функції для реалізації симуляції жителя всередині офісу

Таким чином усього за допомогою декількох ігрових вузлів та функцій коду було реалізовано основний функціонал об’єкта «Житель».

2. Ігрова карта

Ігрова карта, як сцена, також має у своїй основі вузол типу Node2D, просто для логічної організації у ньому усіх інших вузлів.

Також вона містить декілька пустих вузлів того ж типу, необхідних для того, щоб до них додавати в подальшому або просто зберігати ігрові об’єкти різних типів, як у папках – жителів, житлові будинки та офіси.

Основним же вузлом для роботи системи карти є вузол типу NavigationRegion2D – вузол, що місить у собі полігон у вигляді області, по якій можуть пересуватися жителі та використовується ними безпосередньо для навігації та пересування по ігровій карті [19].

Також тут присутні допоміжні вузли типу Label – просто текстові поля для виводу на екрані певних параметрів, як то ФПС в грі. При цьому Label, що відповідає за виведення на екран внутрішньоігрового часу, також служить і для вміщення скрипта, що забезпечує саму систему роботи внутрішньоігрового часу. Окремих скриптів для роботи самої ігрової карти не потрібно

3. Внутрішньоігрова система часу

Як вже було сказано раніше, внутрішньоігрова система часу реалізована не в рамках окремої сцени, а просто вузлом для виводу часу на екран в основній сцені гри, до якого, втім, прикріплено відповідний скрипт.

```
extends Label

var start_time_hours = 7
var start_time_minutes = 30
var current_time_hours = start_time_hours
var current_time_minutes =start_time_minutes
var time_scale = 2

signal time_signal(hours)

func _process(delta):
    current_time_minutes += delta * time_scale
    if(current_time_minutes >= 60):
        current_time_hours += 1
        current_time_minutes -= 60
        time_signal.emit(current_time_hours)
    set_text(str(current_time_hours) + ":" + str(floor(current_time_minutes)))
```

Рисунок 12 – Реалізація внутрішньоігрової системи часу

Цей скрипт є доволі простим, але, тим не менш, ефективним – на початку задаються параметри часу в симуляції, як то час початку відліку та поточних час в годинах і хвилинах, а також «масштаб» часу, що дозволяє гнучко прискорювати або сповільнювати симуляцію, а також сигнал, що сповіщає інші об’єкти про зміну часу. Після цього, в момент початку симуляції, задається стартовий час в грі, а потім він постійно змінюється, видаючи сигнал для інших об’єктів кожну годину.

3. Житлова будівля

Основою цього об’єкта також служить вузол Node2D, що виконує роль контейнера для інших вузлів, а також визначає положення об’єкта в просторі на ігровій карті.

Окрім того, тут присутній ще один вузол типу Node2D, положення якого використовується як позначення місця, де знаходиться вхід у будівлю, адже логічно він не співпадає з центром/положенням самої будівлі.

Також використовується вузол Timer для відліку часу в деяких процесах – зокрема, при покиданні будівлі жителями, що відбувається не одночасно.

Допоміжний вузол типу Label використовується просто для відображення поточної кількості жителів у будівлі.

До кореневого вузла Node2D прикріплено скрипт, що і відповідає за роботу усіх систем цього об’єкта. Перш за все, відбувається ініціалізація усіх необхідних змінних, зокрема, сцени самого жителя, якого будинок створює на ігровій карті:

```
extends Node2D

var citizen = preload("res://Unit.tscn")
var living = []# 100

@onready var entrance_point = get_node("EntrancePoint")
@onready var text_label = get_node("Label")

@onready var offices =  $"../../Offices".get_children()
@onready var port =  $"../../Port"
@onready var time_system =  $"../../TimeLabel"
```

Рисунок 13 – Ініціалізація параметрів ігрового об’єкту «Житлова будівля»

На початку гри цей об’єкт створює задану кількість жителів, генеруючи їм випадкові параметри, після чого кожен з них починає симулюватися самостійно, вже за власним скриптом:

```
func _ready():
    randomize()
    for i in range(100):
        var rand_index = randi() % offices.size()
        var new_citizen_data = {"living": self, "working": offices[rand_index]}
        var new_citizen = citizen.instantiate()
        new_citizen.home = self
        new_citizen.position = entrance_point.global_position
        new_citizen.status = "staying_home"
        new_citizen.workplace = new_citizen_data["working"]
        new_citizen.destination = new_citizen.workplace
        get_parent().get_parent().find_child("Characters").add_child(new_citizen)
```

Рисунок 14 – Генерація жителів об’єктом «Житлова будівля»

Також присутня невелика функція для постійного відображення кількості поточних жителів у будівлі:

```
func _process(delta):
    text_label.set_text(str(living))
```

Рисунок 15 – Відображення об’єктом «Житлова будівля» поточної кількості жителів в ньому

5. Офісна будівля

Структура офісної будівлі загалом теж є дуже схожою до житлового будинку, адже в поточній реалізації, їх функціонал та механіки, з технічної точки зору, не сильно то й відрізняються.

Кореневий вузол типу Node2D для організації усіх інших вузлів, визначення положення об’єкта у просторі та вміщення на собі скрипта з кодом.

Додатковий вузол типу Node2D, положення якого використовується як позначення місця, де знаходиться вхід у будівлю для взаємодії з нею жителів.

Вузол Timer для відліку часу в деяких процесах – зокрема, при покиданні будівлі жителями, що відбувається не одночасно.

Допоміжний вузол типу Label для відображення поточної кількості жителів у будівлі.

Скрипт також є загалом дуже подібним і відрізняється лише назвами деяких змінних для більш логічного їх опису, а також відсутністю функції для генерації жителів, а натомість має параметри, що описують робочий процес, такі як кількість робочих годин на день (умовна) та зарплатня на годину.

В цілому ж усі змінні є майже тими самими, без особливих відмінностей:

```
extends Node2D

@onready var workers_label = get_node("WorkersLabel")
@onready var entrance_point = get_node("EntrancePoint")

var workers = 0
var working_hours = 8
var hour_salary = 100
```

Рисунок 16 – Ініціалізація параметрів ігрового об’єкту «Офісна будівля»

```
func _process(delta):
    workers_label.set_text(str(workers))
```

Рисунок 17 – Відображення об’єктом «Офісна будівля» поточної кількості працівників в ньому

Окрім того, необхідно створити окрему сцену що відповідатиме області всередині офісної будівлі і де проводитиметься симуляція жителів що працюють там в поточний момент:

Кореневий вузол типу Node2D для організації усіх інших вузлів, визначення положення об’єкта у просторі та вміщення на собі скрипта з кодом.

Додатковий вузол типу Node2D, положення якого використовується як позначення місця, де знаходиться вихід з будівлі для взаємодії з нею жителів.

Вузол типу NavigationRegion2D для навігації жителів всередині будівлі.

4.2. Реалізація цільної симуляції та її тестування

Маючи усі повністю реалізовані ігрові об’єкти по окремості, не складає великої складнощі зібрати їх у цільну симуляцію, що буде коректно працювати належним чином.

В сцені гри створюємо вузол типу Sprite, куди передаємо зображення ігрової карти. Змінюємо область навігації вузла NavigationRegion2D таким чином, щоб вона відповідала зображенню. Налаштовуємо систему часу та текстові поля для відображення внутрішньоігрового часу і ФПС у грі, для зручності поставивши їх у верхній правий і лівий кути області відображення. Для більш гнучкого налаштування цієї самої області також додатково створюємо вузол типу Camera2D, що відповідає за відображення зображення на екрані, робимо його активним та налаштовуємо.

Тепер додамо усі необхідні об’єкти для початку симуляції, а саме житлові будинки та офіси, розставивши їх по своїм місцям та налаштувавши початкову кількість жителів в них. Самих жителів додавати немає необхідності, вони будуть створені автоматично скриптами будівель, коли прийде відповідний для цього час.

Також, для зручності, створимо в сцені три дочірні вузли типу Node2D, що будуть просто слугувати «папками» для об’єктів житлових будівель, офісів та жителів, щоб не заповнювати їми основну структуру сцени та спростити орієнтацію по ній, а також налаштуємо ці об’єкти відповідним чином для коректної роботи в такій системі. В результаті структура сцени гри буде виглядати наступним чином:

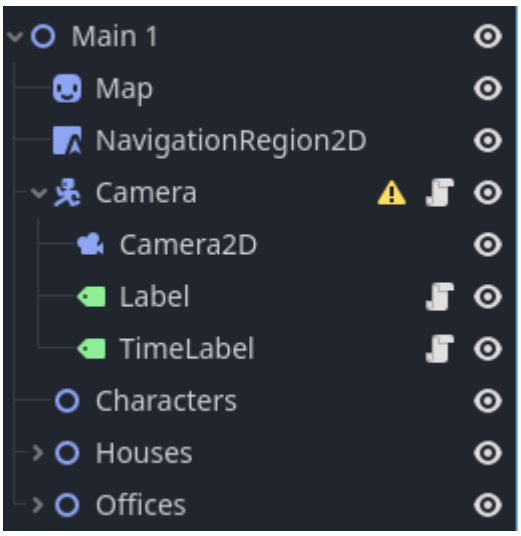

Рисунок 18 – Структура головної сцени гри

А сама ж ігрова сцена, в редакторі рушія, з увімкненим відображенням області навігації, буде виглядати так:

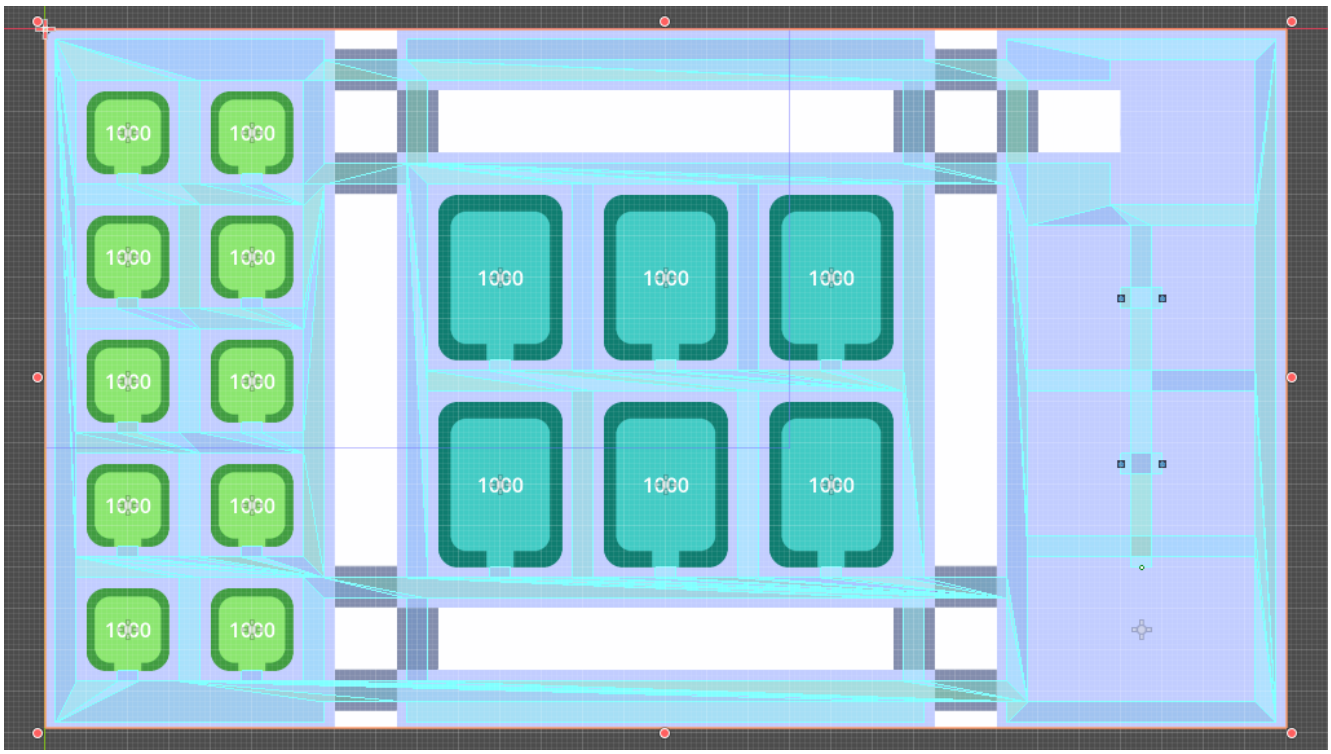

Рисунок 19 – Вигляд сцени гри у редакторі рушія

Налаштувавши усе таким чином, запустимо симуляцію та будемо спостерігати за нею деякий час, щоб переконатися, що все працює належним чином та усі ігрові об’єкти діють строго згідно з задуманої для них логіки та механік.

Як і було задумано за сценарієм, вранці, о 8 годині за внутрішньоігровим часом, усі жителі поступово полишають свої будинки і рухаються до свого місця

роботи. Потім, о 9 годині вони виходять з роботи і повертаються до своїх домівок. Цей цикл повторюється, відповідно, кожні дві години. Такий малий проміжок часу на «роботу» та «сон» був обраний для того, щоб продемонструвати коректність саме ігрової логіки, не витрачаючи при цьому зайвий час на години «застою». Як можемо бачити, система працює цілком коректно, і жителі кожного разу виходять саме на свою роботу, а потім повертаються саме до своїх будинків, що каже про відсутність логічних проблем та дозволяє перейти до наступних етапів роботи.

Рисунок 20 – Демонстрація симуляції при піковому навантаженні

А ось як виглядатиме офісна будівля зсередини в середині робочого дня:

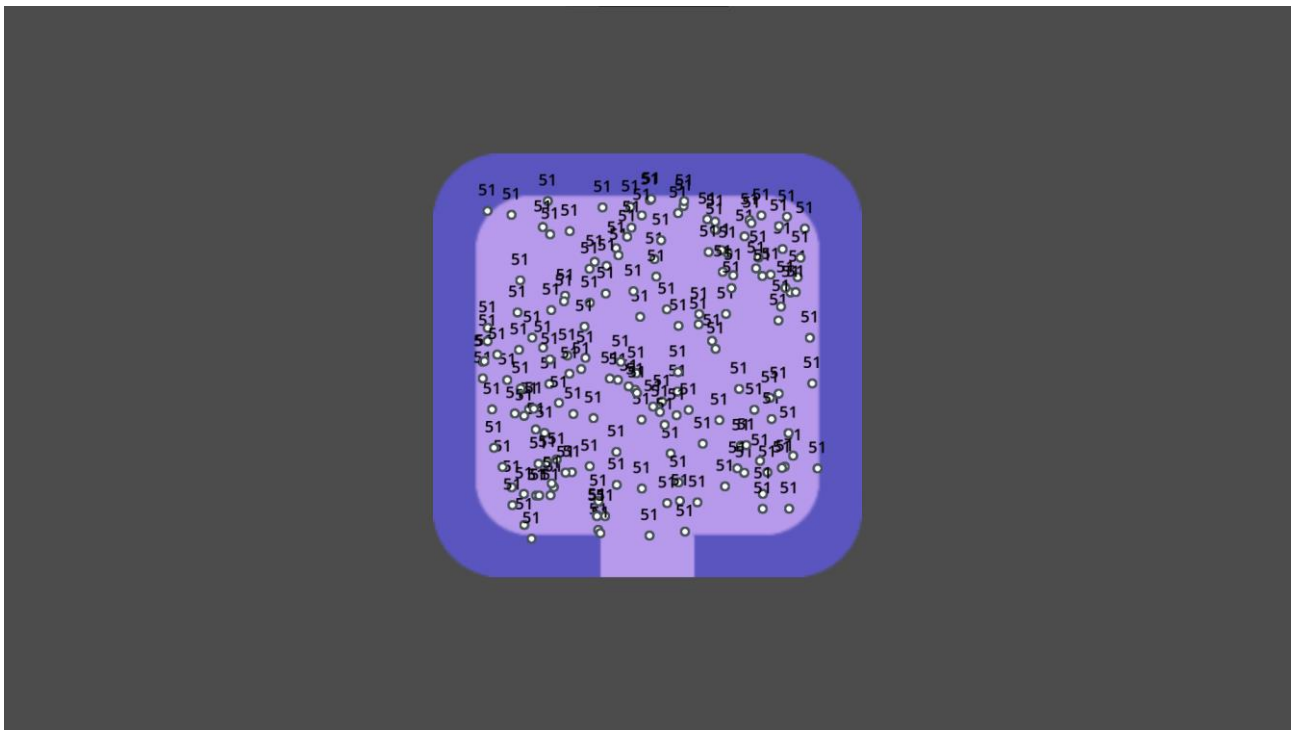

Рисунок 21 – Симуляція жителів всередині офісної будівлі

Проведемо тепер тестування продуктивності симуляції за визначеними раніше процедурами та параметрами. Його результати зведемо у таблицю. Але перш за все, як і було заздалегідь заплановано, необхідно протестувати рушій гри на наявність в ньому оптимізацій типу «Багатопотоковість» та «Оптимізований рендеринг». При цьому варто пам’ятати, що оптимізація «Проста графіка» вже є впровадженою в гру за замовчуванням.

Ознайомившись з документацією та більш детально вивчивши розділ Налаштувань, було виявлено, що в рушії є можливість виставати для гри обмеження по кількості доступних для використання потоків системи, або ж зробити доступними усі відразу, незалежно від їх кількісті, встановивши параметр кількості доступних потоків рівний «-1». Враховуючи це, при тестуванні до впровадження методів оптимізації кількість доступних потоків буде встановлена як 1, а при тестування після їх впровадження – як усі доступні потоки (4 на тестовому ПК). Ручна ж реалізація багатопотоковості є складним і довгим процесом, а також навряд-чи дасть більший ефект, ніж вже впроваджена в рушій, то ж ця ідея розглядатися та виконуватися не буде

Також, написавши простий скрипт для руху камери за допомогою клавіш та рухаючи область відображення за межі зони симуляції і карти, було виявлено, що в рушії відсутня оптимізація «Оптимізований рендериг». З описаних вище причин, реалізація такої оптимізації напряму в грі є складною і наврядчи буде ефективної, а редагування вихідного коду самого рушія також є вкрай складним процесом. Тому, з цих причин, дана оптимізація не буде впроваджена в проект і обидва тестування будуть відбуватися без неї.

Таким чином, результуючі дані тестування за описаних вище умов до впровадження визначених методів оптимізації можна знайти у таблиці 2.

Таблиця 2 – Результати тестування неоптимізованої версії гри

Час тестування в симуляції	**ФПС в грі, од.**	**Споживання ЦП, %**	**Споживання ГПУ, %**	**Споживання ОЗУ, МБ**
Початок симуляції з сплячими жителями	43	80	33	350
Максимальне навантаження жителями	30	85	35	350
Середина робочого дня	40	80	34	350
Вечірнє максимальне навантаження	45	85	34	360

Більш конкретний висновок про ці результати можна буде дати вже після впровадження визначених методів оптимізації та порівняльного тестування, але загалом можна сказати, що вони не зовсім задовільні, оскільки при пікових навантаження ФПС в грі падає нижче 60.

4.3. Впровадження визначених методів оптимізації в гру та повторне тестування

Тепер же, маючи повністю функціонуючу згідно з задуманої логіки гру, впровадимо в неї визначені раніше методи оптимізації та проведемо порівняльну характеристику двох версій гри з точки зору їх продуктивності.

Перш за все, як і було сказано раніше, необхідно врахувати те, що метод оптимізації «Проста графіка» присутній і в не оптимізованій версії гри через його технічні особливості.

Метод «Оптимізований рендериг» же навпаки, через його технічні особливості, не буде реалізований в оптимізованій версії гри.

Для методу «Багатопотоковість», як вже було сказано раніше, просто буде встановлено налаштування у самому рушії, на дозвіл використовувати усі доступні потоки процесора (в даному випадку – чотири).

Також, як вже теж було сказано вище, метод «Випадкові величини та події» для своєї реалізації потребує неоптимізовану версію якоїсь доволі складної механіки, реалізація якої займе доволі багато часу, і тому, в даній роботі не буде реалізований.

Інші методи можуть бути реалізовані без проблем, тож перейдемо до їх безпосередньої реалізації.

1. Часткова симуляція

Цей метод, з точки зору логіки та за кількістю необхідних до внесення змін в ігровий код є найбільш складним та комплексним, але разом з тим дає і найбільш ефективні результати.

Його конкретна технічна реалізація буде полягати в тому, що тепер, підходячи до будівлі, житель не буде продовжувати симулюватися, просто буцімто увійшовши до неї, а повністю зникатиме з ігрової карти, а замість цього до параметрів будівлі в окремий спеціальний масив будуть заноситися усі його дані, а також змінюватимуться відразу усі змінні, відповідно до симуляції, наприклад, виплачуватиметься зарплатня на роботі в офісі. Після завершення

часу перебування в будівлі, вона, на основі записаних раніше даних, буде створювати такого самого жителя і його симуляція продовжиться.

Для цього, перш за все, змінимо код будівель. Зміни для житлових і офісних будівель будуть фактично однаковими, тож розглянемо їх на одному прикладі.

Змінимо тип змінної, що раніше зберігала кількість поточних жителів в будівлі, на масив:

```
var living = []
```

Рисунок 22 – Зміна типу змінної для зберігання жителів в коді будівель

При початку гри будемо не відразу ж створювати жителів на ігровій карті, а лише зберігати в цьому масиві їх дані, до моменту, поки не буде потрібно створити їх на ігровій карті:

```
func _ready():
    for i in range(100):
        randomize()
        var rand_index = randi() % offices.size()
        living.push_front({"living": self, "working": offices[rand_index], "money": 0})
    time_system.time_signal.connect(_on_time_label_time_signal)
```

Рисунок 23 – Зміна коду для генерації жителів в житловій будівлі

Підключимо систему часу та створимо додатковий вузол типу Timer для створення жителів на карті в потрібний час:

```
func _on_time_label_time_signal(hours):
    if(hours % 2 == 0):
        $Timer.start()
```

```
func _on_timer_timeout():
    if living.size() == 0:
        $Timer.stop()
        return
    var new_citizen_data = living.pop_front()
    var new_citizen = citizen.instantiate()
    new_citizen.home = self
    new_citizen.position = entrance_point.global_position
    get_parent().get_parent().find_child("Characters").add_child(new_citizen)
    new_citizen.status = "going_to_work"
    new_citizen.workplace = new_citizen_data["working"]
    new_citizen.destination = new_citizen.workplace
    new_citizen.money = new_citizen_data.money
```

Рисунок 24 – Код для створення жителів на ігровій карті житловою будівлею в потрібний момент з затримками

І на останнє, додамо функцію, яку зможе викликати житель, щоб «зайти» у будівлю, будучи занесеним до відповідного масиву та припинивши свою симуляцію:

```
func catch_citizen(citizen):
    var new_sitizen_data = {"living": citizen.home, "working": citizen.workplace,  "money": citizen.money}
    living.push_front(new_sitizen_data)
```

Рисунок 25 – Функція для занесення жителя в будівлю

Всі інші функції і механіки ігрового об’єкта житель працюють так само, як і раніше.

Тепер перейдемо до коду жителя, де також необхідно внести деякі значні зміни.

Перш за все, оскільки жителю вже немає потреби очікувати кінця робочого дня або сну самому, то відповідні методи **_on_timer_timeout()** та **_on_time_label_time_signal(hours)** а також методи **on_path_finighed()** та **get_salary()** можна просто видалити, залишивши лише ті методи, що відповідають за навігацію по ігровій карті.

По-друге, тепер, при досягненні своєї цілі, житель буде не просто збільшувати кількість жителів в ній на 1, а потім, при покиданні, на -1, а буде викликати функцію для запису себе у масив поточних жителів у будівлі, а потім зникатиме з ігрової карти:

```
    if(finished == true):
        if(status == "going_to_work"):
            status = "working"
            money += workplace.working_hours * workplace.hour_salary
            destination.catch_citizen(self)
            queue_free()
        elif(status == "going_to_home"):
            status = "sleeping"
            destination.catch_citizen(self)
            queue_free()
```

Рисунок 26 – Оновлений код для взаємодії жителя з будівлями

Всі інші функції і механіки ігрового об’єкта житель працюють так само, як і раніше.

На цьому роботу над оптимізацією «Часткова симуляція» можна вважати успішно завершеною і переходити до наступного пункту роботи».

2. Просунуте кешування

Реалізація даного методу, з технічної точки зору, є набагато більш простою, ніж попереднього, та потребує зміни усього лише декількох рядків коду, але, тим не менш, є вкрай важливою з точки зору оптимізації.

Як вже було сказано раніше, даний метод розглядатиметься на прикладі отримання працівником своєї зарплатні за кожен робочий день. Маючи в параметрах офісу зарплатню за кожну годину роботи та кількість робочих годин день, замість того, щоб кожного робочого дня при виплати зарплатні кожному працівникові, перемножувати ці два значення, зробимо це лише один раз, на початку гри, а потім збережемо результат отриманої операції і будемо просто використовувати його. Таким чином ми позбавимося від необхідності кожного ігрового дня, при піковому навантаженні, виконувати тисячу операцій множення двох чисел з плаваючою комою, що є доволі непоганим результатом для економії потужностей ЦП.

Тож вираховуємо загодя необхідний результат у коді офісу:

```
func _ready():
    time_system.time_signal.connect(_on_time_label_time_signal)
    salary_day = working_hours * hour_salary
```

Рисунок 27 – Попереднє обрахування зарплатні за день в коді офісної будівлі

Та будем використовувати його у коді робітника замість постійного обчислення зарплатні за день:

```
    if(finished == true):
        if(status == "going_to_work"):
            status = "working"
            money += workplace.salary_day
            destination.catch_citizen(self)
            queue_free()
```

Рисунок 28 – Оновлений код для зарахування зарплатні жителю

І на цьому, в рамках обраної симуляції, даний метод реалізації можна вважати цілком успішно реалізованим.

На цьому усі обрані методі оптимізації були, по можливості, впроваджені в гру, тож можна тепер перейти безпосередньо до тестування вже оптимізованої версії гри. Результати, знову ж таки, занесемо до таблиці 3.

Таблиця 3 – Результати тестування оптимізованої версії гри

Час тестування в симуляції	**ФПС в грі, од.**	**Споживання ЦП, %**	**Споживання ГПУ, %**	**Споживання ОЗУ, МБ**
Початок симуляції з сплячими жителями	60	3	40	175
Максимальне навантаження жителями	60	40	40	335
Середина робочого дня	60	5	40	335
Вечірнє максимальне навантаження	60	40	45	450

Отримані дані попередньо дають зробити висновки про те, що впроваджені методи оптимізації дійсно дали результат у вигляді покращення продуктивності гри. Більш детальний же аналіз буде проведено далі.

4.4 Впровадження в гру експериментального методу оптимізації та його тестування

Як вже було сказано на початку роботи, основна увага буде приділена саме новому, експериментальному методу оптимізація «Фрагментована симуляція», тому він буде реалізований в окремій версії гри, замість методу «Часткова симуляція» (дані методи хоч і є частково сумісними, але в даному випадку це не має сенсу, оскільки є лише один тип об’єктів, всередині яких можлива ця симуляція, а окрім того, мета даної частини роботи – порівняння цих двох методів) разом з іншими методами, що були реалізовані у попередньому пункті роботи.

Оскільки в жителя в будівлі є усього дві механіки – рух у випадкову точку і отримання грошей, просимулюємо їх наступним чином:

1. Кінцева точка руху визначається кожні декілька секунд випадковим чином. Маючи поточний час і час початку роботи можемо провести генерацію необхідну кількість разів та відразу отримати кінцевий результат. А щоб вона була точною і відповідала задуманій, при генерації кожного разу застосовуватимемо унікальний для кожного жителя ключ – сід.
2. Гроші жителя на поточний момент визначимо просто помноживши час з початку робочого дня на зарплатню на годину:

```
func fragment_simulate(work_start_time):
    var framgent_simulation_time = (time_system.current_time_hours - work_start_time) * 60
    framgent_simulation_time += time_system.current_time_minutes

    for i in range (framgent_simulation_time/5):
        var working_destination = Vector2(randf_range(-256, 256), randf_range(-512, 0))
        destination = working_destination

    money += framgent_simulation_time * workplace.hour_salary / 60
```

Рисунок 29 – Реалізація «Фрагментованої симуляції»

Простестувавши гру, пересвідчимось, що логіка гри працює цілком вірно і усі ігрові об’єкти діють згідно з запланованими для них механіками, після чого можемо переходити до тестування продуктивності у грі. Результати тестування занесено у Таблицю 4:

Таблиця 4 – Результати тестування експериментально оптимізованої версії гри

Час тестування в симуляції	**ФПС в грі, од.**	**Споживання ЦП, %**	**Споживання ГПУ, %**	**Споживання ОЗП, МБ**
Початок симуляції з сплячими жителями	60	5	40	275
Максимальне навантаження жителями	60	40	40	355
Середина робочого дня	60	5	40	355
Вечірнє максимальне навантаження	60	40	45	475

На основі цих даних можна зробити висновок, що таких підхід до оптимізації гри дозволяє зробити гру значно більш реалістичної та глибокою, даючи грацеві відчуття того, що симуляція дійсно проходить усюди і в повному об’ємі, при цьому показники продуктивності є не гіршими, ніж при застосуванні методу «Часткова симуляція», який такої переваги не має, є саме: трохи більше споживання ресурсу ЦП (на 2%) в деяких моментах симуляції, та вище в середньому на 50 МБ споживання ОЗП впродовж усієї симуляції, що є природнім, адже тепер ігрові об’єкти мають у собі більше змінних та параметрів.

4.5. Висновки до розділу

В цьому розділі була виконана, без перебільшення, основна частина роботи усього дипломного проекту, а саме – успішно створено прототип самої гри-симуляції та впроваджено в нього визначені методи оптимізації, що дало змогу провести тестування двох версій гри, зафіксувавши та порівнявши їх показники по цілому ряду параметрів.

Перш за все, важливим результатом стало те, що показник ФПС в грі внаслідок впроваджених методів оптимізації піднявся фактично в півтора-два рази та тримався на рівні 60 впродовж усього тестування, не просідаючи навіть у найбільш активні моменти симуляції, а це є чудовим абсолютним показником, оскільки співпадає з частотою оновлення більшості сучасних моніторів та є певним стандартом у ігровій індустрії. До оптимізації ж цей показник складав 30-45 одиниць, в залежності від моменту симуляції, що є візуально помітним для гравця та викликає неприємний ефект заторможування.

Аналіз інших параметрів дає змогу зрозуміти, чим була викликана така незадовільна продуктивність в неоптимізованій версії грі та чому в оптимізованій ця ситуація різко змінилася на краще. Завантаження ЦП в неоптимізованій версії гри складало 80%-85%, в залежності від стадії симуляції. Якщо врахувати ще і інші процеси, що відбувалися у цей час у системі тестового ПК, то навантаження фактично складало усі 100%, що й спричиняло падіння ФПС, оскільки ЦП просто не встигав оброблювати усі необхідні дані для кожного кадру. В оптимізованій же версії гри цей показник складає 40% у найбільш навантажені моменти, коли велика кількість жителів йде по вулицях міста і небзпосередньо симулюється та усього лише 3%-5% коли усі жителі міста сплять або працюють, тобто можна сказати, що в цей момент вони фактично зовсім не навантажують ПК. А оскільки результати симуляції у обох варіантах фактично є однаковими, то це дійсно говорить про чудову оптимізацію, заради якої не доводиться жертвувати логікою симуляції, або графічною складовою гри.

Споживання ж потужностей ГПУ у оптимізованій версії гри навпаки є більшим – 40%-45% проти 33%-35% у неоптимізованій версії гри. Але і такому, на перший погляд, погіршенню продуктивності, є логічне пояснення. В неоптимізованій версії гри основним обмеженням була потужність процесора, який встигав обробляти дані лише для 30-45 кадрів на секунду, після чого відеокарта відмальовувала ці кадра на моніторі комп'ютера. В оптимізованій же версії грі процесор встигає обробити усі 60 кадрів, а значить і відеокарті, відповідно, доводиться відмальовувати їх усі, що, логічно, збільшує навантаження на неї. Тож в даному випадку це не є наслідком того, що впроваджені методи оптимізації якимось чином погіршили продуктивність гри з точки зору споживання ресурсів відеокарти, і, відповідно, не може вважатися негативним ефектом від них.

Споживання ОЗУ в неоптимізованій версії гри склало приблизно 350 МБ у всіх сценаріях, в той час як в оптимізованій версії гри цей показник приймав значення у доволі широкому діапазоні – від початку 175 МБ на початку симуляції до 450 МБ у її кінці. Але загалом, в середньому, показники були такими ж, або навіть трохи менше, ніж в неоптимізованій версії гри (335 МБ проти 350 МБ), а отже тут також немає погіршення продуктивності від впроваджених методів оптимізації.

Підсумовуючи вищесказане можна зробити висновок, що впроваджені методи оптимізації дійсно ефективно покращили продуктивність гри, особливо там, де це було найбільш необхідно, значно (в 2-28 разів, в залежності від стадії симуляції) зменшивши навантаження на ЦП та дозволивши досягти показника у стабільні 60 ФПС з великим запасом продуктивності (споживання лише 40% ЦП), у противагу 30-45 ФПС у неоптимізованій версії гри.

Основним же результатом роботи можна вважати реалізацію експериментального методу оптимізації «Фрагментована симуляція», який на практиці показав кращі результати ніж його попередники, зробивши симуляцію значно більш реалістичною для гравця і при цьому не завдавши значної шкоди продуктивності.

5. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

Проведемо оцінку основних функціональних та вартісних характеристик програмного продукту, розробленого для оптимізації симуляції міста у комп'ютерних іграх.

Даний програмний продукт призначений для використання на персональних комп'ютерах та під управлінням операційних систем Windows та Linux.

Виконаємо аналіз різних варіантів реалізації модулю інтеграції з метою вибору оптимальної стратегії створення програмного продукту, враховуючи економічні фактори та характеристики продукту, що впливають на продуктивність роботи і на його сумісність з апаратним та програмним забезпеченням. Для цього використаємо апарат функціонально- вартісного аналізу.

Функціонально-вартісний аналіз – це технологія, за допомогою якої виконується оцінка вартості продукту або послуги незалежно від організаційної структури компанії або підприємства. Потрібний обсяг ресурсів на кожному етапі виробництва визначає прямі та побічні витрати, які розподіляються по продуктам та послугам. Дії, що виконуються на цих етапах, називаються функціями у контексті метода ФВА.

Метою ФВА є забезпечення найбільш оптимального розподілу ресурсів, які використовуються для виробництва продукції або надання послуг, а також на прямі та непрямі витрати. У даному випадку програмного продукту необхідно виконати аналіз його функцій й виявити усі витрати на реалізацію цих функцій.

На початку необхідно визначити послідовності функцій, необхідних для виробництва продукту. Виконується перелік усіх можливих функцій, які розподіляються на дві групи: ті, що впливають на вартість кінцевого продукту і ті, що не будуть впливати. Також потрібно оптимізувати цю послідовність скорочуючи кроки, які не будуть впливати на витрати.

Визначається повний обсяг витрат за рік та робочі часи для кожної функції. Потім визначається кількісна характеристика джерел витрат на основі оцінок витрат функцій. Після визначення джерел витрат потрібно провести кінцевий розрахунок витрат необхідних для виробництва продукту.

5.1. Постановка задачі техніко-економічного аналізу

Метод ФВА був застосований для проведення техніко-економічний аналізу розробки комп’ютерної гри, що демонструє ефективність впроваджених методів оптимізації симуляції міста. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для роботи на ПК користувача.

Маємо такі технічні вимоги до програмного продукту:

1. можливість виконання на операційній системі Windows;
2. наявність сумісності;
3. зручність для користувача;
4. можливість налаштування;
5. мінімальні витрати для експлуатації даного продукту.

5.1.1. Обґрунтування функцій програмного продукту

Головна функція F0 – розробка програмного продукту, який демонструє ефективність впроваджених методів оптимізації в комп’ютерну гру з симуляції міста. Виходячи з даної мети, виділимо такі основні функції програмного продукту:

1) F1 – вибір ігрового рушія;

2) F2 – вибір мови програмування;

3) F3 – вибір ПК для тестування;

4) F4 – вибір системи контролю версій.

Визначені основні функції можна реалізувати різними способами.

Варіанти функції F1:

а) Godot

б) GameMaker

в) Unity

Варіанти функції F2:

а) Python

б) Java

в) C#

Варіанти функції F3:

а) Ноутбук розробника

б) ПК розробника

Варіанти функції F4:

а) Без СКВ

б) Git\Github

в) PlasticSCM

5.1.2. Варіанти реалізації основних функцій

Морфологічна карта системи на рисунку 5.1 показує варіанти реалізації основних функцій. Всі можливі комбінації варіантів реалізації цих функцій складають повну множину варіантів для даного програмного продукту.

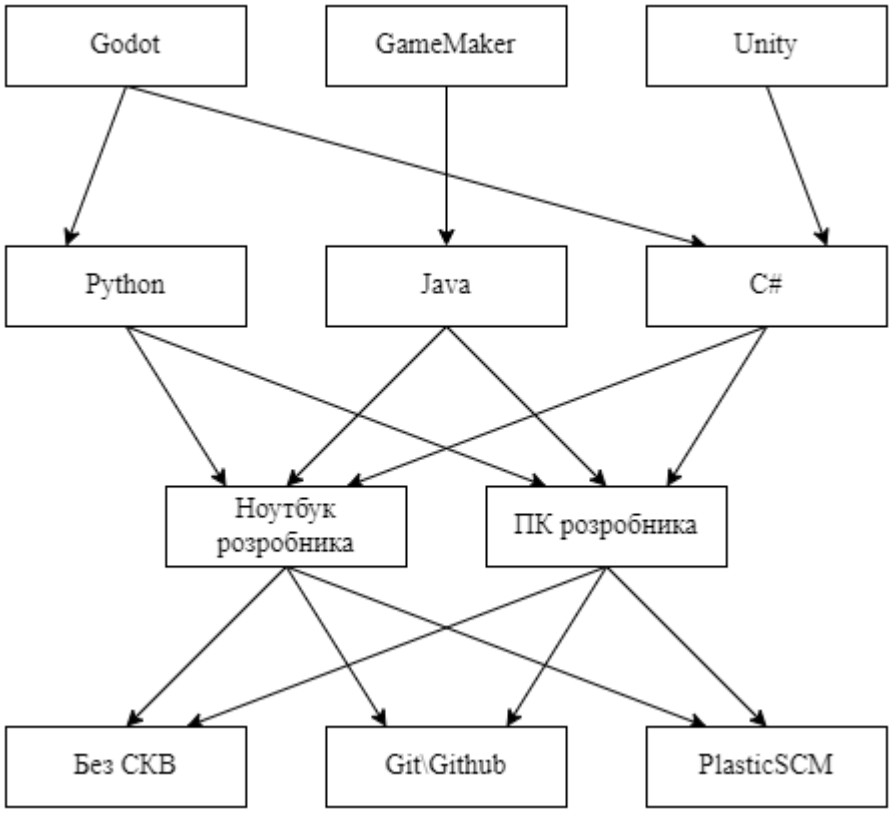

Рисунок 30 – Морфологічна карта системи

Побудуємо позитивно-негативну матрицю варіантів основних функцій на основі морфологічної карти програмного продукту (табл. 5).

Таблиця 5 – Позитивно-негативна матриця

Функція	Варіант реалізації	Переваги	Недоліки
F1	А	Простота в освоєнні, відносно висока продуктивність роботи	Мала кількість навчального матеріалу, проблеми з адаптацією гри до деяких платформ
	Б	Простота в освоєнні, висока продуктивність, багатий інструментарій	Мала кількість навчального матеріалу, відсутність підтримки 3Д
	В	Потужність та обширність інструментарію, наявність великої бібліотеки розширень та плагінів	Проблеми з продуктивністю, відносна складність освоєння
F2	А	Простота в освоєнні та розробці, велика кількість бібліотек	Низька продуктивність, відсутність інструментарію низького рівня
	Б	Висока продуктивність, універсальність, кросплатформенність	Складний для освоєння синтаксис, платна ліцензія для комерційної розробки
	В	Висока продуктивність, потужний інструментарій, широкий спектр застосування	Складність з адаптацією до різних платформ, проблеми з підтримкою старих версій
F3	А	Мобільність, можливість тестування будь-де та відразу в ході розробки	Обмежена потужність, можлива нестача продуктивності

Продовження таблиці 5

F3	Б	Потужність, можливість тестування більш навантажених симуляцій	Обмежена мобільність, необхідній перенесення гри для тестування
F4	А	Простота, відсутність додаткових дій по роботі з СКВ	Складність в веденні розробки, відсутність історії змін
	Б	Швидка синхронізація, гнучке налаштування	Складність в використанні без додаткових інструментів
	В	Потужність у великих проектах, зручний інтерфейс та система роботи	Необхідність реєстрації у системі, створення кабінету організації для проекту, тощо.

В результаті аналізу позитивно-негативної матриці потрібно відкинути деякі варіанти реалізації функцій, оскільки вони не відповідають поставленим рперед програмним продуктом задачам.

Аналіз функції F1:

Оскільки необхідно реалізувати проект у стиснуті строки, то має сенс використовувати ті рушії, з якими найбільше знайомий розробник, щоб зменшити затрати на освоєння, тому можна відкинути варіант Б.

Аналіз функції F2:

Додаткові витрати під час розробки, зокрема, на платні ліцензії, є вкрай небажаними, тож можемо відкинути варіант Б.

Аналіз функції F3:

Тестування на менших потужностях дозволяє більш наочно показати результати оптимізації, тож залишимо варіант А.

Аналіз функції F4:

В даному випадку проект є доволі простим за своєю структурою та не містить великих файлів, тож відпадає потреба у варіант В.

В результаті маємо такі варіації для розробки програмного продукту:

1) F1(А) – F2(А) – F3(А) – F4(А)

2) F1(А) – F2(А) – F3(А) – F4(Б)

3) F1(А) – F2(В) – F3(А) – F4(А)

4) F1(А) – F2(В) – F3(А) – F4(Б)

5) F1(В) – F2(В) – F3(А) – F4(А)

6) F1(В) – F2(В) – F3(А) – F4(Б)

Необхідно виконати оцінку якості розглянутих функцій. Для цього оберемо систему параметрів.

5.2 Обґрунтування системи параметрів програмного продукту

5.2.1 Опис параметрів

Використовуючи дані про основні функції програмного продукту, визначаються основні параметри, що будуть використані для розрахунку коефіцієнта технічного рівня. Використаємо наступні параметри для опису характеристик продукту:

1) X1 – споживання ресурсів відеокарти симуляцією
2) X2 – споживання ресурсів ЦП симуляцією;
3) X3 – швидкодія роботи симуляції;;
4) X4 – кількість рядків коду, що треба написати розробнику;

5.2.2. Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів обираємо на основі вимог замовника й потреб експлуатації ПП. Ці значення занесено у таблицю 6.

Таблиця 6 – Основні параметри програмного продукту

Опис параметру	Умовні позначення	Одиниці виміру	Значення параметра		
			Гірші	Середні	Кращі
Споживання ресурсів відеокарти симуляцією	X1	Відсотки	50	25	10
Споживання ресурсів ЦП симуляцією	X2	Відсотки в середньому на потік	100	75	50
Швидкодія роботи симуляції	X3	Кадри	30	45	60
Кількість рядків коду, що треба написати розробнику	X4	Рядки	1000	500	250

За даними таблиці 4 будуються графіки значень параметрів (див. рис. 28-31).

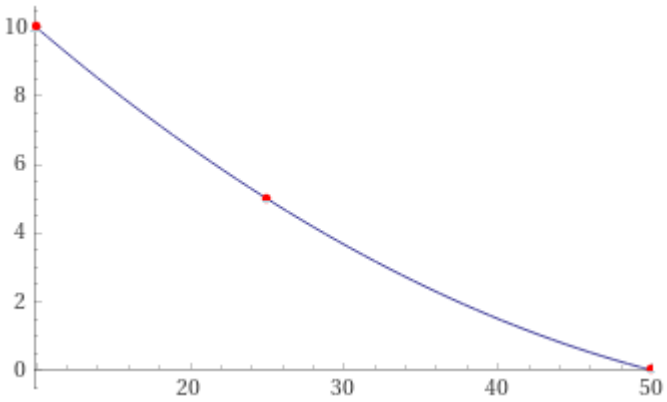

Рисунок 31 – Споживання ресурсів відеокарти

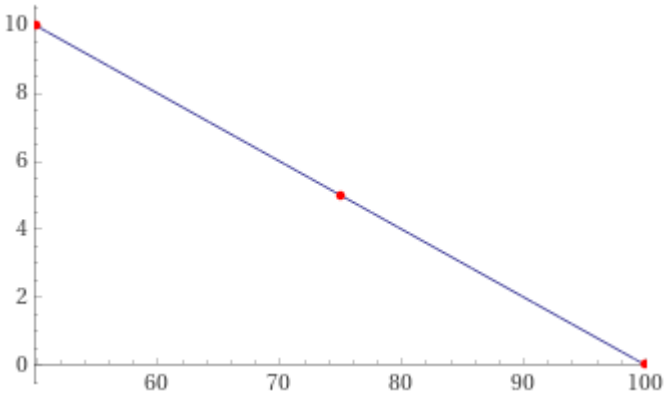

Рисунок 32 – Споживання ресурсів ЦП

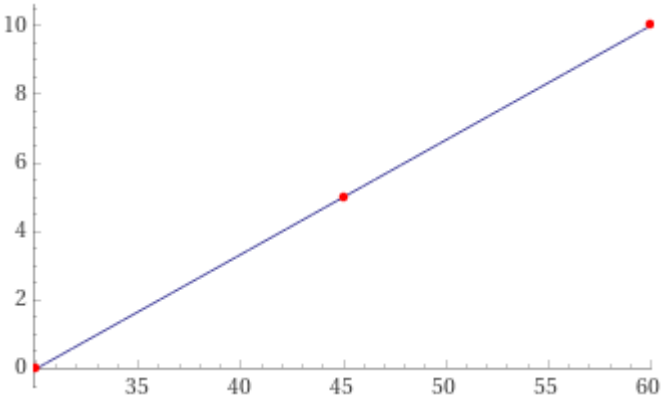

Рисунок 33 – Продуктивність симуляції

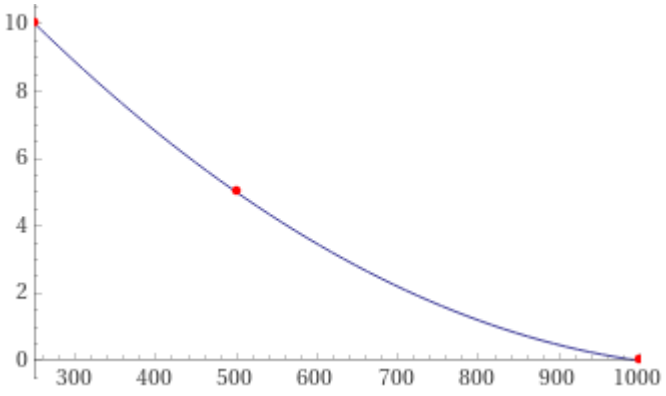

Рисунок 34 – Об’єм коду, що необхідно написати програмісту

5.2.3. Аналіз експертного оцінювання параметрів

Далі необхідно виконати обговорення та аналіз кожного параметру для конкретно поставленої цілі. Кожен експерт оцінює ступінь важливості кожного параметру для розробки програмного продукту, який має найбільше значення для кінцевого результату.

Визначаємо значення кожного параметра за методом попарного порівняння. Для проведення експертної комісії необхідно 5 людей. Для оцінки коефіцієнтів значимості необхідно:

1) присвоїти різні ранги в залежності від рівня значимості параметрів;

2) перевірити придатність експертних оцінок;

3) виконати попарну оцінку параметрів;

4) обробити результати та розрахувати коефіцієнт значимості.

Підсумки експертного ранжування подано у таблиці 5.

Таблиця 7 – Підсумки експертного ранжування

Параметр	Ранг параметра за експертною оцінкою					Сума рангів	Відхилення Δ_i	Δ_i^2
	1	2	3	4	5			
X1	1	1	2	1	2	6	-5.5	30.25
X2	4	3	3	4	4	18	+5.5	30.25
X3	2	4	4	3	3	16	+3.5	12.25
X4	3	2	1	2	1	9	-3.5	12.25
Разом	10	10	10	10	10	50	0	85

Щоб перевірити ступінь достовірності експертних оцінок, необхідно визначити наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} * r_{ij} = 50,$$

де r_{ij} – ранг i-го параметра, визначений j-м експертом, N – число експертів.

б) середня сума рангів T:

$$T = \frac{1}{n} R_i = 12{,}5.$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T.$$

г) загальна сума квадратів відхилення:

$$S = \sum_{j=1}^{n} * \Delta_i^2 = 85.$$

д) коефіцієнт узгодженості (конкордації):

$$W = \frac{12 * S}{N^2(n^3 - n)} = \frac{12 * 85}{5^2(4^3 - 4)} = 0.68 > W_k = 0{,}67.$$

Ранжирування вважається достовірним, оскільки розрахований коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67. Проведемо попарне порівняння всіх параметрів за результатами ранжирування. Результати попарного порівняння занесемо у таблицю 8. Числове значення, що визначає ступінь переваги і-го параметра над j-тим, aij визначається за формулою:

$$a_{ij} = \begin{cases} 1{,}5 & x_i > x_j; \\ 1{,}0 & x_i = x_j; \\ 0{,}5 & x_i < x_j. \end{cases}$$

Таблиця 8 – Результати ранжування параметрів

Параметри	Експерти					Підсумкова оцінка	Числове значення коефіцієнтів переваги
	1	2	3	4	5		
X1, X2	<	<	<	<	<	<	0,5
X1, X3	<	<	<	<	<	<	0,5
X1, X4	<	<	>	<	>	<	0,5
X2, X3	>	<	<	>	>	>	1,5
X2, X4	>	>	>	>	>	>	1,5
X3, X4	<	>	>	>	>	>	1,5

Отримані числові оцінки утворюють матрицю $A = \|a_{ij}\|$. Для кожного параметра розрахунок вагомості K_{B_i} проводиться за наступною формулою:

$$K_{B_i} = \frac{b_i}{\sum_{i=1}^{n} b_i}$$

Де $b_i = \sum_{j=1}^{N} a_{ij}$ – вага i-го параметра за результатами оцінок усіх експертів; a_{ij} – коефіцієнт переваги i-го параметра над j-м параметром.

Потрібно розраховувати відносні оцінки декілька разів, поки різниця між 70 наступними та попередніми значеннями не стане відрізнятися від попередніх менше ніж на 2%. На другому та подальших кроках відносні оцінки розраховуються за наступною формулою:

$$K_{B_i} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}$$

Де $b_i' = \sum_{j=1}^{N} a_{ij}\, b_i$.

За розрахунками, різниця значень коефіцієнтів вагомості після четвертої ітерації не перевищує 2% (табл. 9), тому подальші ітерації не потрібні.

Таблиця 9 – Розрахунок вагомості параметрів

i	j				Перша ітерація		Друга ітерація		Третя ітерація		Четверта ітерація	
	X1	X2	X3	X4	b_i	K_{B_i}	b_i	K_{B_i}	b_i	K_{B_i}	b_i	K_{B_i}
X1	1,0	0,5	0,5	0,5	2,5	0,16	9,25	0,16	34,13	0,16	125,06	0,16
X2	1,5	1,0	1,5	1,5	5,5	0,34	21,25	0,36	77,88	0,36	285,06	0,36
X3	1,5	0,5	1,0	1,5	4,5	0,28	16,25	0,28	59,13	0,27	216,56	0,27
X4	1,5	0,5	0,5	1,0	3,5	0,22	12,25	0,21	44,88	0,21	164,56	0,21
Разом					16,00	1,00	59,00	1,00	216,00	1,00	791,25	1,00

5.3. Аналіз рівня якості варіантів реалізації функцій

На даному етапі потрібно визначити рівень якості виконання основних функцій для кожного варіанту.

Абсолютне значення параметра X1 (споживання ресурсів відеокарти) було обрано середнім, для задоволення вимог по підтримці.

Абсолютне значення параметра X2 (споживання ресурсів ЦП) обрано найкращим, для задоволення вимог продуктивності даного програмного продукту.

Абсолютне значення параметра X3 (продуктивність симуляції) обрано не найгіршим. Це значення буде відповідати або варіанту 60 ФПС, або варіанту 45 ФПС.

Абсолютне значення параметра X4 (об'єм коду, що необхідно написати програмісту) обрано не найгіршим, тоді це значення буде 500 або 250 рядків.

Для визначення коефіцієнту технічного рівня якості для кожного варіанту, виконаємо розрахунки за формулою:

$$K_{TP} = \sum_{i=1}^{n} K_{B_i} B_i,$$

де n – кількість параметрів; K_{B_i} – коефіцієнт вагомості i-го параметра; B_i – оцінка i-го параметра в балах.

Розрахунок показників рівня якості наведено в таблиці 10.

Таблиця 10 – розрахунок показників якості

Основні функції	Варіант реалізації	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	А	25	5	0,16	0,8
F2	А	50	10	0,36	3,6
F3	А	60	10	0,27	2,7
	Б	45	5		1,35
F4	А	250	10	0,21	2,1
	Б	500	5		1,05

Визначимо рівень якості кожного з варіантів за даними з таблиці 9:

1) F1(А) – F2(А) – F3(А) – F4(А) = 0,8 + 3,6 + 2,7 + 2,1 = 9,2

2) F1(А) – F2(А) – F3(А) – F4(Б) = 0,8 + 3,6 + 2,7 + 1,05 = 8,15

3) F1(А) – F2(А) – F3(Б) – F4(А) = 0,8 + 3,6 + 1,35 + 1,05 = 6,8

4) F1(А) – F2(А) – F3(Б) – F4(Б) = 0,8 + 3,6 + 1,35 + 2,1 = 7,85

Отже, з наявних варіантів найкращим є перший варіант. Коефіцієнт технічного рівня буде мати найбільше значення для нього.

5.4 Економічний аналіз вартості розробки програмного продукту

Визначимо вартість розробки ПП, для чого спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе одне окреме завдання:

1) Розробка логічної частини програмного продукту;

2) Розробка ігрової частини програмного продукту

Завдання 1 та 2 за ступенем новизни відноситься до групи А. За складністю алгоритми, який використовуються в обох завданнях належать до групи 3. В ході реалізації завдання використовують інформацію у вигляді перемінних даних, наданих користувачем. Необхідно провести розрахунок норм часу на розробку та програмування для завдання. Визначимо загальну трудомісткість за формулою.

$$ТО = ТР \cdot КП \cdot КСК \cdot КМ \cdot КСТ \cdot КСТ.М$$

де ТР – трудомісткість розробки програмного продукту; КП – поправочний коефіцієнт; КСК – коефіцієнт на складність вхідної інформації; КМ – коефіцієнт рівня мови програмування; КСТ – коефіцієнт використання стандартних модулів і прикладних програм; КСТ.М – коефіцієнт стандартного математичного забезпечення.

Виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 3, маємо, що для першого завдання трудомісткість дорівнює ТР = 21 людино-день. Поправочний коефіцієнт, яким враховується вид вхідної інформації для цього завдання: КП = 1,95. Поправочний коефіцієнт, яким враховано складність контролю вхідної та вихідної інформації для цього завдання, рівний КСК = 1. Коефіцієнт КСТ залишаємо 1, оскільки стандартні модулі при розробці не використовуються. При роботі над першим завданням використовується мова Python, тому врахуємо поправковий коефіцієнт КМ = 1,1. Розрахуємо загальну трудомісткість програмування для першого завдання:

T1 = 21*1,95*1,1 = 45,045 людино-днів

Аналогічно розрахуємо загальну трудомісткість для другого завдання, в якому використовується алгоритми третьої групи складності зі ступенем новизни А. Тоді ТР = 14 людино-днів, КП = 1,75, КСК = 1, КСТ = 1, КМ = 1,1:

T1 = 14*1,75*1,1 = 26,95 людино-днів

Загальна трудомісткість усіх варіантів реалізації збігається, тому їх можна об'єднати в одну групу. Тоді, загальна трудомісткість складає:

ТО = (45,045 + 26,95) · 8 = 575,96 людино-годин;

В розробці беруть участь один програміст з окладом 10 000 гривень та один аналітик з окладом 17 500 тисяч гривень. За даною формулою визначимо середню зарплату за годину:

$$С_ч = \frac{M}{T_m * t} \text{ грн.,}$$

де M – місячний оклад працівників; T_m – кількість робочих днів на місяць; t – кількість робочих годин в день.

$$С_ч = \frac{10\,000 + 17\,500}{2 * 20 * 8} = 85{,}9375 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за формулою:

$$С_{ЗП} = С_ч * T_i * К_д$$

де $С_ч$ – величина погодинної оплати праці програміста; T_i – трудомісткість відповідного завдання; $К_д$ – норматив, який враховує додаткову заробітну плату.

Визначимо зарплату розробників за варіантами:

$$С_{ЗП} = 85{,}94 * 575{,}96 * 1{,}2 = 59\,396{,}60 \text{ грн.}$$

Відрахування на соціальний внесок становить 22%:

$$С_{СВ} = С_{ЗП} * 0{,}22 = 13\,067{,}25 \text{ грн.}$$

Також необхідно розрахувати витрати на оплату однієї машино-години. Одна ЕОМ обслуговується одним інженером апаратного забезпечення з окладом 12 500 грн. та коефіцієнтом зайнятості КЗ = 0,25. Таким чином для однієї машини отримаємо:

$$С_г = 12 * М * К_З = 12 * 12\,500 * 0{,}25 = 37\,500 \text{ грн.}$$

Врахуємо додаткову заробітну плату:

$$С_{ЗП} = С_г * (1 + К_З) = 37\,500 * (1 + 0{,}25) = 46\,875 \text{ грн.}$$

Відрахування на соціальний внесок становить 22%:

$$С_{СВ} = С_{ЗП} * 0{,}22 = 46\,875 * 0{,}22 = 10\,312{,}5 \text{ грн.}$$

Розрахуємо амортизаційні відрахування за формулою при амортизації 25% на рік та при вартості ЕОМ – 47 500 грн.:

$$С_А = К_{ТМ} * К_А * Ц_{ПР} = 1{,}25 * 0{,}25 * 47\,500 = 14\,843{,}75 \text{ грн.}$$

де $К_{ТМ}$ – коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача; $К_А$ – річна норма амортизації; $Ц_{ПР}$ – договірна ціна приладу.

Розрахуємо витрати на ремонт та профілактику за формулою:

$$С_Р = К_{ТМ} * К_Р * Ц_{ПР} = 1{,}25 * 0{,}075 * 47\,500 = 4\,453{,}13 \text{ грн.}$$

де $К_Р$ – відсоток витрат на поточні ремонти.

Розраховуємо ефективний годинний фонд часу ПК на рік за формулою:

$$Т_{ЕФ} = (Д_К - Д_В - Д_С - Д_Р) * t * К_В$$

$$Т_{ЕФ} = (365 - 100 - 11 - 20) * 8 * 0{,}8 = 1\,497{,}6 \text{ год.}$$

де $Д_К$ – календарна кількість днів у році; $Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів; $Д_Р$ – кількість днів планових ремонтів устаткування; t – кількість робочих годин в день; $К_В$ – коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} * N_C * К_З * Ц_{ЕЛ} = 1497{,}6 * 0{,}75 * 0{,}45 * 4{,}87 = 2\,461{,}49 \text{ грн.},$$

де N_C – середньо-споживча потужність приладу; $К_З$ – коефіцієнт зайнятості приладу; $Ц_{ЕЛ}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} * 0{,}67 = 47\,500 * 0{,}67 = 31\,825 \text{ грн.}$$

Розрахуємо річні експлуатаційні витрати, які будуть складати:

$$С_{ЕК} = С_{ЗП} + С_{СВ} + С_А + С_Р + С_{ЕЛ} + С_Н$$

$$С_{ЕК} = 46\,875 + 10\,312{,}5 + 14\,843{,}75 + 4\,453{,}13 + 2\,461{,}49 + 31\,825 =$$

$$= 110\,770.87 \text{ грн.}$$

Тоді собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{МГ} = \frac{С_{ЕК}}{С_{ЕФ}} = \frac{110\ 770{,}87}{1\ 497{,}6} = 73{,}97\ грн/год.$$

Враховуючи те, що в даному випадку всі роботи, що пов‘язані з розробкою програмного продукту проводяться на ЕОМ, витрати на оплату часу її роботи складають:

$$С_{М} = С_{МГ} * Т = 73{,}97 * 575{,}96 = 42\ 603{,}76\ грн.$$

Тоді накладні витрати складають 67% від заробітної плати:

$$С_{Н} = С_{ЗП} * 0{,}67 = 59\ 396{,}60 * 0{,}67 = 39\ 795{,}72\ грн.$$

Отже, вартість розробки програмного продукту за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{СВ} + С_{М} + С_{Н}$$

$$С_{ПП} = 59\ 396{,}60 + 10\ 312{,}5 + 42\ 603{,}76 + 39\ 795{,}72 = 152\ 108{,}58\ грн.$$

Оскільки вартість розробки для всіх варіантів реалізації не відрізняється, розрахуємо коефіцієнт техніко-економічного рівня для найкращого варіанта (з найбільшим коефіцієнтом технічного рівня) за формулою:

$$К_{ТЕР} = \frac{К_{К}}{С_{ПП}} = \frac{9{,}2}{152\ 108{,}58} = 6{,}05 * 10^{-5}$$

6. ВИСНОВКИ

В результаті виконання даної дипломної роботи було проведено дослідження методів оптимізації симуляції міста в комп’ютерних іграх, вивчено приклади реальних комп’ютерних ігор, що мають подібні механіки, та теоретично можливі до впровадження методи оптимізації.

Було сформовано методику дослідження методів та тестування отриманих результатів, за допомогою базових примітивів – об’єктів та їх взаємодії між собою розглянуто симуляцію міста з логічної точки зору, та знайдено механіки, в яких було можливо впровадження тих чи інших методів оптимізації.

На основі отриманих даних, та з урахуванням технічних можливостей рушія і самої гри досліджені методи були успішно впроваджені, після чого, шляхом порівняння заздалегідь визначених параметрів для оцінки продуктивності, будо доведено, що дані методи дійсно є ефективними, підвищуючи продуктивність гри до якісно кращого рівня.

Було розроблено, досліджено та впроваджено цілком новий метод оптимізації – «Фрагментована симуляція», що показав значно кращі результати, ніж його попередники, адже при близьких до них показників продуктивності дозволив не жертвувати значною долею реалістичності симуляції та не обмежувати гравця в його діях щодо дослідження світу гри.

Також в ході роботи було виконано функціонально-вартісний аналіз програмного продукту, в результаті чого було визначено найкращий варіант реалізації даного проекту та його коефіцієнт техніко-економічного рівня, який складає $К_{ТЕР} = 6{,}05 * 10^{-5}$.

7. СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. SimCity (1989 video game) : [Електронний ресурс] // Вікіпедія – вільна енциклопедія. – Режим доступу до ресурсу:
SimCity (1989 video game) - Wikipedia – дата звернення 08.06.2023.
2. GTA 5 : [Електронний ресурс] // Вікіпедія – вільна енциклопедія. – Режим доступу до ресурсу:
Grand Theft Auto V - Wikipedia – дата звернення 08.06.2023.
3. Скріншоти гри GTA V : [Електронний ресурс] // Сторінка гри на офіційному сайті розробника – Режим доступу до ресурсу:
Screens from Grand Theft Auto V for PC - Rockstar Games – дата звернення 08.06.2023.
4. Cities Skylines Citizens : [Електронний ресурс] // Skylines Wiki – підтримувана фанатами енциклопедія по грі Cities Skylines. – Режим доступу до ресурсу:
Citizens — Cities: Skylines Wiki (paradoxwikis.com) – дата звернення 08.06.2023.
5. Скріншоти гри Cities Skylines : [Електронний ресурс] // Сторінка гри на цифровому майданчику Steam – Режим доступу до ресурсу:
Cities: Skylines y Steam – дата звернення 08.06.2023.
6. Особливості гри Dwarf Fortress : [Електронний ресурс] // Сторінка гри на офіційному сайті розробника – Режим доступу до ресурсу:
Bay 12 Games: Dwarf Fortress – дата звернення 08.06.2023.
7. Скріншоти гри Dwarf Fortress : [Електронний ресурс] // Сторінка гри на офіційному сайті розробника – Режим доступу до ресурсу:
Bay 12 Games: Dwarf Fortress – дата звернення 08.06.2023.
8. Багатопотоковість (комп’ютерна архітектура) : [Електронний ресурс] // Вікіпедія – вільна енциклопедія. – Режим доступу до ресурсу:
Multithreading (computer architecture) - Wikipedia – дата звернення 08.06.2023.
9. Кешування в коп’ютерних іграх : [Електронний ресурс] // – Режим доступу до ресурсу:
Leveraging the Power of Cache Memory – дата звернення 08.06.2023.

10. Оптимізований рендеринг : [Електронний ресурс] // Вікіпедія – вільна енциклопедія. – Режим доступу до ресурсу:
Відкладене освітлення і затінювання – дата звернення 08.06.2023.
11. Особливості ігрового рушія Godot Engine : [Електронний ресурс] // Офіційний сайт рушія Godot Engine. – Режим доступу до ресурсу:
Features - Godot Engine – дата звернення 08.06.2023.
12. Підтримувані мови програмування ігрового рушія Godot Engine : [Електронний ресурс] // Офіційна документація рушія Godot Engine. – Режим доступу до ресурсу:
Scripting languages — Godot Engine documentation – дата звернення 08.06.2023.
13. Сцени та вузли : [Електронний ресурс] // Офіційна документація рушія Godot Engine. – Режим доступу до ресурсу:
Scenes and nodes — Godot Engine documentation – дата звернення 08.06.2023.
14. Вузол Node2D : [Електронний ресурс] // Офіційна документація рушія Godot Engine. – Режим доступу до ресурсу:
Node2D — Godot Engine documentation – дата звернення 08.06.2023.
15. Вузол Sprite2D : [Електронний ресурс] // Офіційна документація рушія Godot Engine. – Режим доступу до ресурсу:
Sprite2D — Godot Engine documentation – дата звернення 08.06.2023.
16. Вузол NavigationAgent2D : [Електронний ресурс] // Офіційна документація рушія Godot Engine. – Режим доступу до ресурсу:
NavigationAgent2D — Godot Engine documentation – дата звернення 08.06.2023.
17. Вузол Timer : [Електронний ресурс] // Офіційна документація рушія Godot Engine. – Режим доступу до ресурсу:
Timer — Godot Engine documentation – дата звернення 08.06.2023.
18. Вузол Label : [Електронний ресурс] // Офіційна документація рушія Godot Engine. – Режим доступу до ресурсу:
Label — Godot Engine documentation – дата звернення 08.06.2023.

19. Вузол NavigationRegion2D : [Електронний ресурс] // Офіційна документація рушія Godot Engine. – Режим доступу до ресурсу: NavigationRegion2D — Godot Engine documentation – дата звернення 08.06.2023.

8. ДОДАТКИ

Додаток А. Програмний код неоптимізованої версії ігрового об’єкту «Житлова будівля»

```
extends Node2D

var citizen = preload("res://Unoptimized Version/UnoptimisedUnit.tscn")

@onready var offices =  $"../../Offices".get_children()
@onready var time_system =  $"../../Camera/TimeLabel"

@onready var entrance_point = get_node("EntrancePoint")
@onready var text_label = get_node("Label")

var living = 0

func _ready():
	randomize()
	for i in range(100):
		var rand_index = randi() % offices.size()
		var new_citizen_data = {"living": self, "working": offices[rand_index]}
		var new_citizen = citizen.instantiate()
		new_citizen.home = self
		new_citizen.position = entrance_point.global_position
		new_citizen.status = "staying_home"
		new_citizen.workplace = new_citizen_data["working"]
		new_citizen.destination = new_citizen.workplace

	get_parent().get_parent().find_child("Characters").add_child(new_citizen)

func _process(delta):
	text_label.set_text(str(living))
```

Додаток Б. Програмний код неоптимізованої версії ігрового об’єкту «Офісна будівля»

```
extends Node2D

@onready var workers_label = get_node("WorkersLabel")
@onready var entrance_point = get_node("EntrancePoint")

var workers = 0
var working_hours = 8
var hour_salary = 100

func _process(delta):
	workers_label.set_text(str(workers))
```

Додаток В. Програмний код неоптимізованої версії ігрового об’єкту «Житель»

```
extends Node2D

@onready var time_system = $"../../Camera/TimeLabel"
@onready var nav_reg =  $"../../NavigationRegion2D"
@onready var nav = get_node("NavigationAgent2D")
@onready var timer = get_node("Timer")
@onready var money_label = get_node("MoneyLabel")

var speed = 250
var path = []
var map
var pathing_target = Vector2(0, 0)
var money = 0

var finished = false
var status = "staying_home"

var home : Node2D
var workplace : Node2D
var destination : Node2D

func setup_navserver():
	map = NavigationServer2D.map_create()
	NavigationServer2D.map_set_active(map, true)

	var region = NavigationServer2D.region_create()
	NavigationServer2D.region_set_transform(region, Transform2D())
	NavigationServer2D.region_set_map(region, map)

	var navigation_poly = NavigationMesh.new()
	navigation_poly = $"../../NavigationRegion2D".navigation_polygon
	NavigationServer2D.region_set_navigation_polygon(region, navigation_poly)


func _update_navigation_path(start_position, end_position):
	path = NavigationServer2D.map_get_path(map, start_position, end_position, true)
```

```
        path.remove_at(0)
        set_process(true)

func _ready():
        time_system.time_signal.connect(_on_time_label_time_signal)
        call_deferred("setup_navserver")
        home.living += 1

func start_path():
        _update_navigation_path(self.position,
destination.entrance_point.global_position)
        finished = true

func move_along_path(distance):
        var last_point = self.position
        while path.size():
                var distance_between_points = last_point.distance_to(path[0])
                if distance <= distance_between_points:
                        self.position = last_point.lerp(path[0], distance /
distance_between_points)
                        return
                distance -= distance_between_points
                last_point = path[0]
                path.remove_at(0)
        self.position = last_point
        set_process(false)
        if(finished == true):
                if(status == "going_to_work"):
                        status = "working"
                        workplace.workers += 1
                        money += workplace.working_hours * workplace.hour_salary
                        finished = false
                elif(status == "going_to_home"):
                        status = "staying_home"
                        home.living += 1
                        finished = false

func _process(delta):
        var walk_distance = speed * delta
        move_along_path(walk_distance)
        money_label.set_text(str(money))
```

```
func _on_timer_timeout():
	if(status == "staying_home"):
		destination = workplace
		home.living -= 1
		status = "going_to_work"
		finished = true
		start_path()
	if(status == "working"):
		destination = home
		workplace.workers -= 1
		status = "going_to_home"
		finished = true
		start_path()

func _on_time_label_time_signal(hours):
	if hours % 2 == 0:
		randomize()
		timer.wait_time = randf_range(1, 10)
		timer.start()
	if hours % 2 == 1:
		randomize()
		timer.wait_time = randf_range(1, 10)
		timer.start()
```

Додаток Г. Програмний код оптимізованої версії ігрового об’єкту «Житлова будівля»

```
extends Node2D

var citizen = preload("res://Optimized Version/OptimisedUnit.tscn")

@onready var offices =  $"../../Offices".get_children()
@onready var time_system =  $"../../TimeLabel"

@onready var entrance_point = get_node("EntrancePoint")
@onready var text_label = get_node("Label")

var living = []

func _ready():
	for i in range(100):
		randomize()
		var rand_index = randi() % offices.size()
		living.push_front({"living": self, "working": offices[rand_index], "money": 0})
	time_system.time_signal.connect(_on_time_label_time_signal)

func _on_timer_timeout():
	if living.size() == 0:
		$Timer.stop()
		return
	var new_citizen_data = living.pop_front()
	var new_citizen = citizen.instantiate()
	new_citizen.home = self
	new_citizen.position = entrance_point.global_position
	get_parent().get_parent().find_child("Characters").add_child(new_citizen)
	new_citizen.status = "going_to_work"
	new_citizen.workplace = new_citizen_data["working"]
	new_citizen.money = new_citizen_data["money"]
	new_citizen.destination = new_citizen.workplace

func _process(delta):
	text_label.set_text(str(living.size()))
```

```
func _on_time_label_time_signal(hours):
	if(hours % 2 == 0):
		$Timer.start()

func catch_citizen(citizen):
	var new_sitizen_data = {"living": citizen.home, "working": citizen.workplace, "money": citizen.money}
	living.push_front(new_sitizen_data)
```

Додаток Ґ. Програмний код оптимізованої версії ігрового об’єкту «Офісна будівля»

```
extends Node2D

var citizen = preload("res://Optimized Version/OptimisedUnit.tscn")

@onready var homes = $"../../Houses".get_children()
@onready var time_system =  $"../../TimeLabel"

@onready var workers_label = get_node("WorkersLabel")
@onready var entrance_point = get_node("EntrancePoint")

var workers = []
var working_hours = 8
var hour_salary = 100
var salary_day

func _ready():
	time_system.time_signal.connect(_on_time_label_time_signal)
	salary_day = working_hours * hour_salary

func _process(delta):
	workers_label.set_text(str(workers.size()))

func _on_timer_2_timeout():
	if workers.size() == 0:
		$Timer2.stop()
		return
	var new_citizen_data = workers.pop_front()
	var new_citizen = citizen.instantiate()
	new_citizen.position = entrance_point.global_position
	get_parent().get_parent().find_child("Characters").add_child(new_citizen)
	new_citizen.status = "going_to_home"
	new_citizen.home = new_citizen_data["living"]
	new_citizen.workplace = new_citizen_data["working"]
	new_citizen.money = new_citizen_data["money"]
	new_citizen.destination = new_citizen.home
```

```
func _on_time_label_time_signal(hours):
	if(hours % 2 == 1):
		$Timer2.start()

func catch_citizen(citizen):
	var new_sitizen_data = {"living": citizen.home, "working": citizen.workplace, "money": citizen.money}
	workers.push_front(new_sitizen_data)
```

Додаток Д. Програмний код оптимізованої версії ігрового об’єкту «Житель»

```
extends Node2D

@onready var nav_reg =  $"../../NavigationRegion2D"
@onready var nav = get_node("NavigationAgent2D")
@onready var money_label = get_node("MoneyLabel")

var speed = 250
var path = []
var map
var pathing_target = Vector2(0, 0)
var money = 0

var finished = false
var status = "staying_home"

var home : Node2D
var workplace : Node2D
var destination : Node2D

func setup_navserver():
	map = NavigationServer2D.map_create()
	NavigationServer2D.map_set_active(map, true)

	var region = NavigationServer2D.region_create()
	NavigationServer2D.region_set_transform(region, Transform2D())
	NavigationServer2D.region_set_map(region, map)

	var navigation_poly = NavigationMesh.new()
	navigation_poly = $"../../NavigationRegion2D".navigation_polygon
	NavigationServer2D.region_set_navigation_polygon(region, navigation_poly)


func _update_navigation_path(start_position, end_position, ):
	path = NavigationServer2D.map_get_path(map, start_position, end_position, true)
	path.remove_at(0)
	set_process(true)
```

```
func _ready():
	call_deferred("setup_navserver")

func start_path():
	_update_navigation_path(self.position,
destination.entrance_point.global_position)
	finished = true

func move_along_path(distance):
	var last_point = self.position
	while path.size():
		var distance_between_points = last_point.distance_to(path[0])
		if distance <= distance_between_points:
			self.position = last_point.lerp(path[0], distance /
distance_between_points)
			return

		distance -= distance_between_points
		last_point = path[0]
		path.remove_at(0)
	self.position = last_point
	set_process(false)
	if(finished == true):
		if(status == "going_to_work"):
			status = "working"
			money += workplace.salary_day
			destination.catch_citizen(self)
			queue_free()
		elif(status == "going_to_home"):
			status = "sleeping"
			destination.catch_citizen(self)
			queue_free()

func _process(delta):
	var walk_distance = speed * delta
	move_along_path(walk_distance)
	money_label.set_text(str(money))

func _on_timer_timeout():
	start_path()
```