МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Дослідження методів оптимізації програмних систем з великою кількістю об'єктів в Game Dev індустрії

Виконав:

студент 2 курсу групи ІПЗм-21-2

Колесник В.Ю.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо-наукова

Керівник доц. Назаров О.С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. Кафедри _______________ З.В. Дудар

2023 р

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри ___________

(підпис)

«__»_____2023 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Колесник Владиславу Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів оптимізації програмних систем з великою кількістю об'єктів в Game Dev індустрії»

затверджена наказом університету від «29» березня 2023р. № 302 Ст

2. Термін подання студентом роботи до екзаменаційної комісії «12» травня 2023р.

3. Вихідні дані до роботи *результати проведеного експерименту, рішення оптимізаційної задачі, її аналіз, пояснювальна записка.*

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз предметної області і постановка задачі, дослідження файлів ігор із вихідним кодом, методи оптимізації та критерії їх оцінювання, формулювання та вирішення оптимізаційної задачі, аналіз отриманого результату, проведення експериментів.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
	Аналіз предметної галузі	07.02.2023	виконано
	Постановка задаччі	10.02.2023	виконано
	Пошук інформації у файлах ігор	13.02.2023	виконано
	Формування та рішення оптимізаційної задачі	01.03.2023	виконано
	Проведення експерименту	01.04.2023	виконано
	Оформлення тез	05.04.2023	виконано
	Написання пояснювальної записки	10.05.2023	виконано
	Підготовка презентації та доповіді	15.05.2023	виконано
	Нормоконтроль	15.05.2023	виконано
	Рецензування	20.05.2023	виконано
	Занесення диплома в електронний архів	22.05.2023	виконано
	Захист кваліфікаційної роботи	24.05.2023	

Дата видачі завдання 29 березня 2023 р.

Студент ______________ Колесник В.Ю.
(підпис) (прізвище, ініціали)

Керівник роботи ______________ доц. Назаров О.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи, 69 сторінок, 46 рисунків, 7 додатків,9джерел.

GAME DEV, ОБ'ЄКТ, КЛАС, МЕТОДИ ОПТИМІЗАЦІЇ ПРОГРАМНИХ СИСТЕМ, ШТУЧНИЙ ІНТЕЛЕКТ, ОПТИМІЗАЙІЙНА ЗАДАЧА.

Об'єктом дослідження є методи оптимізації програмних систем з великою кількістю об'єктів, що можуть значно уповільнювати їх роботу. Дослідження у більшій мірі стосується Game Dev індустрії та ігор, де проблема особливо чітко відсзначається.

Метою роботи є виявлення причин навантаження технічного забезпечення ПК в процесі роботи різних програм та ігор, знаходження можливих способів оптимізації таких програм як після введення в експлуатацію, так і в процесі розробки.

У результаті виконання кваліфікаційної роботи було проведено дослідження роботи деяких програмних систем та ігор, що мають проблеми із навантаженням ПК, зокрема мають велику кількість об'єктів. Були надані поради щодо уникнення таких проблем в процесі розробки програмного забезпечення та після введення в експлуатацію.

GAME DEV, OBJECT, CLASS, METHODS OF OPTIMIZATION OF SOFTWARE SYSTEMS, ARTIFICIAL INTELLIGENCE, OPTIMIZATION PROBLEM.

The object of research is methods of optimizing software systems with a large number of objects that can significantly slow down their work. The study is more relevant to the Game Dev industry and games, where the problem is particularly pronounced.

The purpose of the work is to identify the reasons for the load on the technical support of the PC during the operation of various programs and games, to find possible ways to optimize such programs both after commissioning and during the development process.

As a result of the qualification work, a study of the operation of some software systems and games that have problems with PC load, in particular, with a large number of objects, was conducted. Advice was provided on how to avoid such problems during software development and after commissioning.

Я,

Колесник Владислав Юрійович
(прізвище, ім'я, по батькові)

студент групи ІПЗм-21-2 здобувач вищої освіти на другому (магістерському) рівні

кафедра програмної інженерії,
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему
Дослідження методів оптимізації програмних систем з великою кількістю об'єктів в Game Dev індустрії,
(назва роботи)

що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

Вступ 8
1. Аналіз предметної галузі 10
   1.1.Технічна передмова 10
   1.2.Аналіз прецедентів в предметній області 12
2. Використання отриманих знань на практиці 39
   2.1.Проведення експерименту 39
3. Пошук можливих рішень 43
   3.1.Встановлення оптимізаційної задачі 43
4. Проблема попередньої роботи 49
   4.1.Аналіз проблеми 49
Висновки 51
Перелік джерел посилання 52
Додаток А Перелік використаних публікацій за науковими напрямами керівника та нуковців кафедри програмної інженерії 54
Додаток Б Звіт результатів перевірки на унікальність тексту 55
Додаток В Слайди презентації 56
Додаток Г Експертний висновок щодо нормоконтролю 63
Додаток Д Тези публікації 64
Додаток Е Посилання на файли рецензії та її підпису із підприємства 68
Додаток Ж Текст рецензії із підприємства 69

ВСТУП

В рамках попередньої кваліфікаційної робити автора було розроблено бойову спроектовано та реалізовано систему автоматичного бою, що включає в себе юнітів (ігрові сутності), їх баланс, сам процес бою та його механіки. В процесі програмної реалізації головною метою було реалізувати всю логіку системи та її механіки, при цьому притримуючись принципів ООП – використання класів, наслідування тощо. Таким чином, з програмної точки зору, бій був, насправді, простою взаємодією об'єктів класів між собою, проте таких об'єктів, кожний із яких відповідав за окремий корабель (юніт), могло бути дійсно багато, сумарно і п'ять мільйонів і дев'ять. Згодом, в процесі тестування ПЗ, було виявлено дуже повільну роботу системи при таких кількостях юнітів, крім того – використання нею аномально великої кількості ресурсів комп'ютера. Тому, постало логічне питання – як з цією проблемою борються аналогічні ПЗ, яка ситуація з ними в інших проектах Game Dev індустрії та що такого аналогічного можна знайти у всій індустрії розробки ПЗ?.

І дійсно – на перший погляд одразу можна зрозуміти, що ситуація з "великою кількістю юнітів" або "об'єктів", говорячи більш загальними термінами, може бути абсолютно нормальною в деяких IT галузях і доволі поширеною, в тому числі, в Game Dev індустрії.

Серед таких прикладів можна виділити Data Science, Big Data та машинне навчання, в ході роботи необхідно буде звернути увагу на такі більш загальні випадки проблеми оптимізації роботи системи з великою кількістю об'єктів.

Що ж до Game Dev індустрії, то кожен, хто хоч трохи цікавиться індустрією комп'ютерних ігор, знає, що одним із найпоширеніших понять та тем при знайомстві з продуктом є "оптимізація". Буквально кожен рік виходять нові ігрові продукти, які потребують все більшої продуктивність персональних комп'ютерів користувачів, і це зовсім не завжди пов'язано з покращеннями графічної складової ігор, стратегії, в яких графічна складова менш "важка", теж легко можуть впоратися із навантаженнями на "залізо" ПК.

Хоча, заради справедливості, слід зауважили, що графічна складова, з програмної точки зору, також працює на основі об’єктів – кожен візуальний об’єкт є також і програмним об’єктом. Звісно, абсолютна більшість таких об’єктів є статичною – каміння, ґрунт, будинки, предмети інтер’єру, скелі, дерева й інші різноманітні елементи оточення. Такі об’єкти не виконують ніяких дій пов’язаних з складними математичними обчисленнями. А от такі графічні об’єкти, що є візуалізацією програмних сутностей, що безпосередню виконують важливі дії та становлять сам процес гри, можуть навантажувати ПК деякими обчисленнями та зберіганні деякої немалої інформації про них.

Основною метою даної роби буде виявлення проблем роботи систем з великою кількістю об’єктів в IT індустрії, а особливо в Game Dev індустрії, з точки зору оптимізації. В процесі необхідно розглянути наглядні приклади таких проблем, виявити причину їх появи та спробувати запропонувати їх рішення.

Результатом роботи мають стати рекомендації щодо оптимізації роботи систем з великою кількістю об’єктів.

1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1. Технічна передмова

Очевидно, що під об'єктом ми розуміємо екземпляр класу, що є важливою частиною системи і безпосередньо представляє об'єкт із реальної предметної області – чи то нейрон нейронної мережі, чи то конкретну людину в деякій системі банку, чи то юніт у грі, які виконують левову долю роботи всієї системи і разом з цим навантажують ПК.

Сам клас, як і його об'єкти, зазвичай складається з двох частин (див. рис. 1.1) – параметрів, які дають йому характеристику, та функціоналу, що визначає які дії може проводити об'єкт.

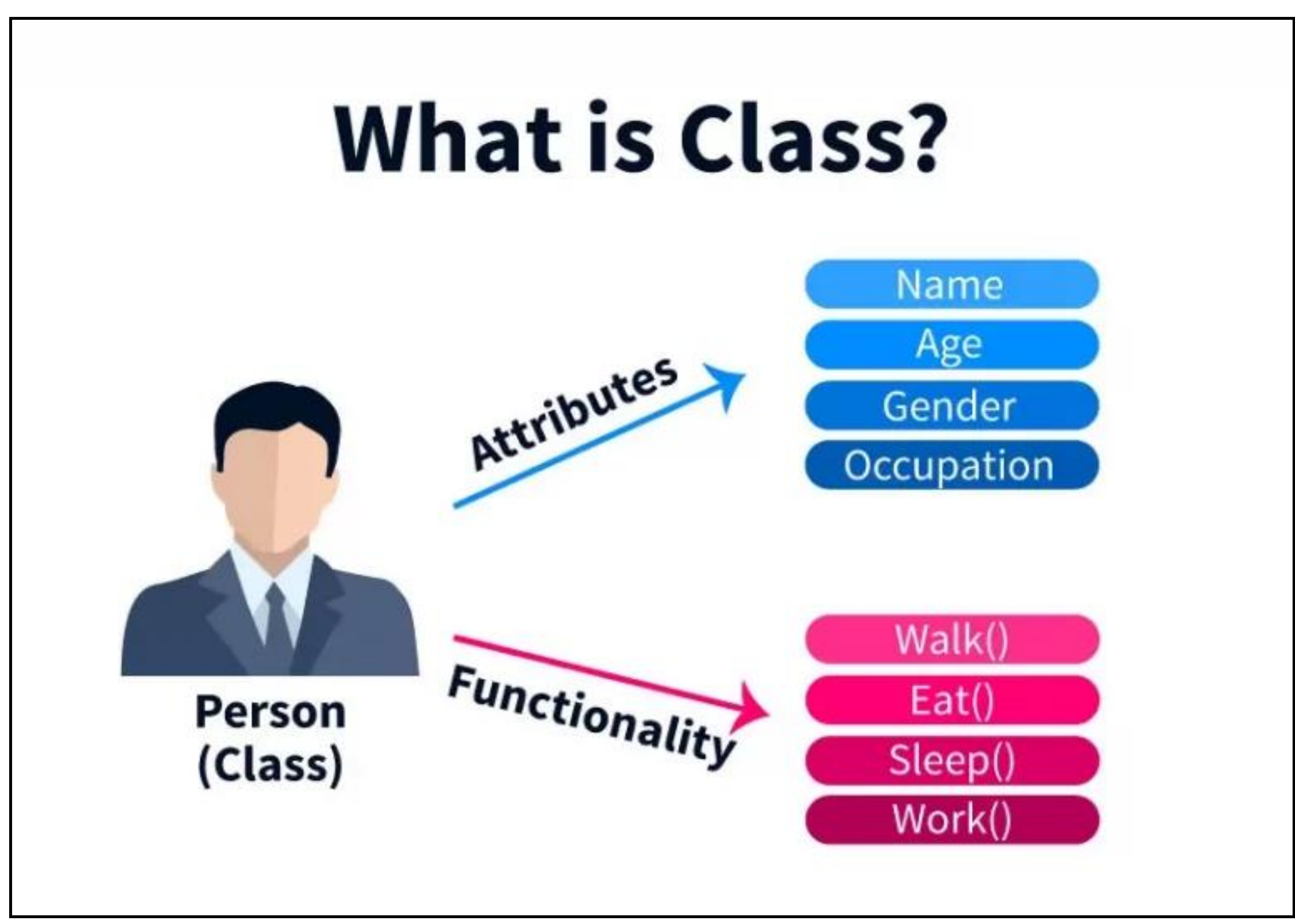

Рисунок 1.1 – Загальна схема класу

Саме ці частини і несуть відповідальність за навантаження системи – будь-яка змінна займає оперативну пам’ять, а функції виконують різноманітні підрахунки та трансформації, словом – навантажують як оперативну пам’ять, адже використовують набір змінних та констант, так і ресурс центрального процесора, який власне і виконує ці дії.

При наявності даних про роботу систем ми будемо намагатися робити деякі розрахунки, аби продемонструвати проблеми що можуть виникнути. Забігаючи наперед – нам знадобляться наступні [1][2]:

- цілочисельний тип даних (int). Змінні цього типу можуть набувати лише цілих значень. Зазвичай тип int займає чотири байти, або 32 біти (2^{32} = 4294967296), від’ємні значення допускаються. Таким чином змінні можуть набувати значень від -2147483648 і до 2147483647. Можна використовувати тип даних uint (unsigned int), він приймає тільки додатні значення та має діапазон від 0 до 4294967295;
- цілочисельний байтовий тип (byte). Виходячи з назви типу він займає в пам'яті один байт, тобто вісім біт. Може містити в собі 2^{8} = 256 значень. Від’ємні значення допускаються, тому тип має діапазон від -128 до 127 (не забуваємо, що є ще число нуль); Іноді byte буває беззнаковий і приймає значення то від 0 до 255, а його знакова версія (sbyte) допускає від’ємні значення;
- короткий цілий тип (short). У пам'яті йому виділено 2 байти, або 16 біт (2^{16} = 65536), може бути від’ємним, приймає значення від -32768 до 32767;
- довгий цілий тип (long). Довгий цілий тип займає у пам'яті 8 байт, або 64 біти. $2^{64} = 1,8446744 \times 10^{19}$. Діапазон допустимих значень дуже великий і допускає від’ємні значення: [-9223372036854775808; 9223372036854775807]. В деяких мовах програмування модифікатор long можна використовувати в поєднанні з іншими типами (long пишеться перед назвою типу, наприклад: long double), розширюючи тим самим діапазон допустимих значень;

- число з плаваючою комою (float). Цей тип також називають речовим типом одинарної точності. float - це ні що інше, як десяткове число (раціональне число), але в пам'яті комп'ютера він представляється у вигляді експоненційного запису: складається з мантиси і показника ступеня. Наприклад: 3,141 = 3141,0 * 10^{-3}, де 3141 - мантиса, а -3 - показник ступеня десяти. Зазвичай займає 4 байти пам'яті;
- число з плаваючою комою подвійної точності (double). Даний тип схожий з типом float, як виходиться із назви – він в два ризи більший, тобто займає 8 байтів пам'яті і, відповідно, має в два рази більший діапазон значень, що приймаються.

1.2. Аналіз прецедентів в предметній області

Почнемо розгляд прикладів із більш загального випадку. Дуже показовою є ситуація при розробці нейронних мереж і штучного інтелекту за допомогою еволюційних алгоритмів.

У штучному інтелекті та машинному навчанні еволюційні алгоритми – це розділ еволюційних обчислень, у яких використовуються моделі процесів природного відбору (розмноження, мутація, рекомбінація та відбір) та принципи природної еволюції для вирішення задач оптимізації [3].

Рішення оптимізаційної задачі розглядаються як особи популяції. Якість кожного рішення оцінюється за допомогою спеціальної функції придатності (fitness function – фітнес-функція), після чого відбувається еволюція популяції. Таким чином, еволюційний алгоритм містить такі кроки:

1) формується початкова популяція шляхом випадкового відбору (перше покоління);
2) оцінюється придатність кожного члена популяції за допомогою фітнес-функції;
3) повторюються такі дії (еволюція):
   - **відбір** - вибір найбільш пристосованих особин для розмноження (батьки);

- **розмноження** - формування нових особин шляхом схрещування та мутації, а потім оцінка їхньої придатності;
- **рекомбінація** - найменш пристосовані особини попереднього покоління замінюються найбільш пристосованими особами нового покоління.

Алгоритм може ставити собі за мету як знаходження єдиної особи в результаті свого виконання, так і групи підходящих осіб. Крім того, зауважимо що найважливішим процесом, з точки зору складності обчислень, є еволюція.

При великих кількостях особин в початковій популяції, можна з впевненістю казати, що процес еволюції займе значно більшу частину алгоритму, ніж підготовчі процеси. Крім того, на складність еволюції впливає складність предметної області, очевидно, що при великій кількості параметрів об’єктів буде більшою кількість обчислень, а у системах, де параметри впливають одне на одного, таке зростання обчислень взагалі буде геометричним.

Припустимо, що кожна нова еволюція схрещує дві особини в одну, тобто з кожною ітерацією особин стає вдвічі менше. Виразимо початкову кількість особин як ступінь двійки, тоді для 2^n особин необхідна кількість об’єднань може бути вирахувана через формулу:

$$\sum_{i=0}^{n-1} 2^i \tag{1}$$

Або спрощено:

$$2^n - 1 \tag{2}$$

Тобто при початковій кількості особин $2^{10} = 1024$ кількість схрещувань буде 1023, враховуючи, що схрещування проводиться між двома особинами маємо 2046 об’єктів схрещування, що означає створення ще 1023 нових особин в процесі еволюції.

На практиці час тренувань штучного інтелекту на базі еволюційного алгоритму може легко займати декілька днів. Врахуємо, що в серйозних проектах задачі такого штучного інтелекту можуть бути надзвичайно складними, що породжує великі витрати ресурсів на розробку математичної моделі, за якою буде працювати алгоритм, а її вдосконалення неможливе без врахування проміжних результатів. Іншими словами – щоб вдосконалити штучний інтелект – треба отримати результат його роботи, що, як було сказано, може займати дні. Таким чином, робимо висновок, що проблема нашого дослідження присутня в цій ситуації.

Тепер торкнемося задач прогнозування та класифікації. Це задачі діагностики і прогнозування деякої величини Y по доступних значеннях змінних $X1,\dots,X$n. Такі задачі часто виникають в різних областях людської діяльності:

- постановка медичного діагнозу або результатів лікування по сукупності клінічних і лабораторних показників;
- прогноз властивостей ще не синтезованої хімічної сполуки по його молекулярній формулі;
- розпізнавання образів;
- діагностика ходу технологічного процесу;
- діагностика стану технічного устаткування;
- прогноз фінансових індикаторів;
- тощо.

Одним із класифікаторів в машинному навчанні, що вирішує такі задачі такого типу, є так званий баєсів класифікатор. Він використовує теорему Баєса для визначення ймовірності приналежності спостереження (елемента вибірки) до одного з класів. Теорема має наступну формулу:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \quad (3)$$

Де A та B є подіями, $P(A)$ та $P(B)$ відповідно їх ймовірностями, $P(A|B)$ – умовна ймовірність A за умови істинності B, $P(B|A)$ – навпаки.

Очевидні наступні твердження:

- в рамках кожного об’єкту, що треба класифікувати, присутні багато параметрів;
- на основі цих параметрів вираховується схожість об’єкту на кожний із класів, що присутній в системі;
- об’єкти не впливають одне на одного;
- при навчанні значення ваг, що визначають належність до кожного класу, буде змінюватися із входженням нових тренувальних об’єктів;
- при вирішенні реальних задач кількість об’єктів може бути безліччю.

Наївний баєсів класифікатор припускає, що параметри незалежні одне від одного, тому, в такому разі, розрахунки при роботі системи будуть досить простими. В рамках класифікації належність до кожного класу можна розрахувати незалежно одне від одного, а отже є можливість використати паралельні обчислення. При навчанні значення ваг змінюються один раз за кожне входження нового об’єкту тренування. Отже, в такому випадку можна сказати, що система доволі гарно оптимізується і проблем з її роботою виникнути не повинно.

Проте, існують не наївні баєсівські класифікатори, які враховують, що параметри об’єкту можуть впливати одне на одного. В такому випадну слід враховувати, що, по-перше, самі обчислення ускладняються через використання принципу нормального розподілу, а, по-друге, важливо розуміти, що реальністю зумовлено те, що деякі параметри реального об’єкту можуть залежати від інших його параметрів, що призведе до додаткових обчислень, що будуть враховувати математичну залежність параметрів одне від одного. (див. рис 1.2).

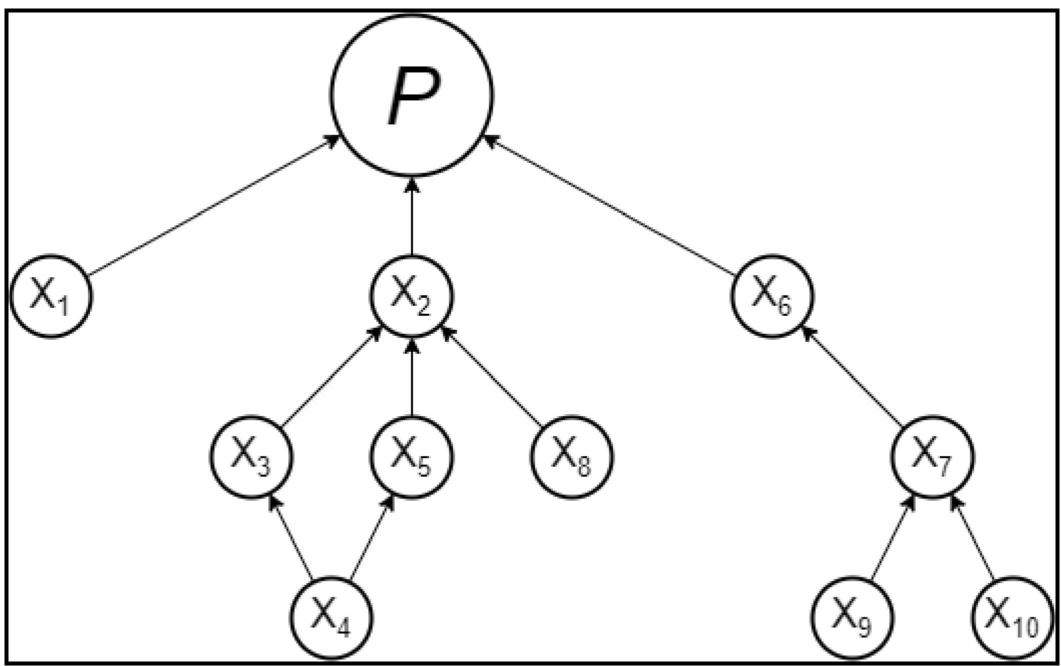

Рисунок 1.2 – Приклад залежності параметрів одне від одного та їх вплив на результат

Таким чином, можна зробити висновок, що у разі залежності параметрів один від одного, система буде працювати більш складно – неможливо одразу визначити належність об’єкту до певного класу, треба поступово вираховувати вплив параметрів одне на одного, а потім і на результат, що унеможливлює такий рівень паралелізму, як при наївному класифікаторі.

Перейдемо до прикладів в Game Dev індустрії. Першою на черзі буде гра “Hearts of Iron IV” – глобальна стратегія за мотивами Другої світової війни. За свої вже 6 років існування гра пройшла багато етапів розробки, які по суті представляли собою великі оновлення, що приносили із собою як нові механіки, так і суттєво змінювали старі. За весь цей час гравці помічали різні проблеми в оптимізації, а ентузіасти знайшли велику кількість різноманітної інформації пов’язаної із вихідним кодом, крім того деякі вказівки на роботу вихідного коду залишили самі розробники. На основі таких даних та власного досвіду виділимо декілька цікавих проблем гри.

Спочатку треба звернути увагу на такий аспект, що пов’язаний безпосередньо із користувацьким інтерфейсом. Однією із складових гри є

повітряна війна, як не складно здогадатися в ній приймають участь літаки, все як в реальності. В рамках гри літаки збираються в крила по 100 літаків для виконання завдань. Також в грі присутні такі сутності як «пілоти-аси», це виправдана реальністю частина ігровою механіки. Аси можуть бути прикріплені до наявних крил та дають їм деякі бонуси (див. рис. 1.3), які конкретно та що вони означають – нам не важливо.

Рисунок 1.3 – Вікно вибору асу для прикріплення до крила, по сумісництву – список всіх наявних асів

В процесі гри кількість літаків може легко досягати більше двох тисяч приблизно всередині ігрової партії, аси також з’являються доволі легко, будь-який гравець, що хоче гарно провести партію, буде намагатися мати їх в такій кількості, аби кожне крило мало закріпленого за ним аса заради тих самих

бонусів. Тому, зазвичай в партіях, що проходять добре для гравця, кількість асів буде дещо більшою за кількість крил.

І тут з'являється перша проблема – в таких ситуаціях, починаючи з двох тисяч літаків, тобто двадцяти крил, кількість асів стає на стільки великою, що процес прикріплення аса до крила починає займати зайвий час через зависання інтерфейсу. Банальне відкриття списку асів для вибору одного може займати декілька секунд, інтерфейс гри просто завмирає, як і сам ігровий процес, а вікно із списком асів тримається напівпрозорим із відсутністю інформації (див. рис. 2.4).

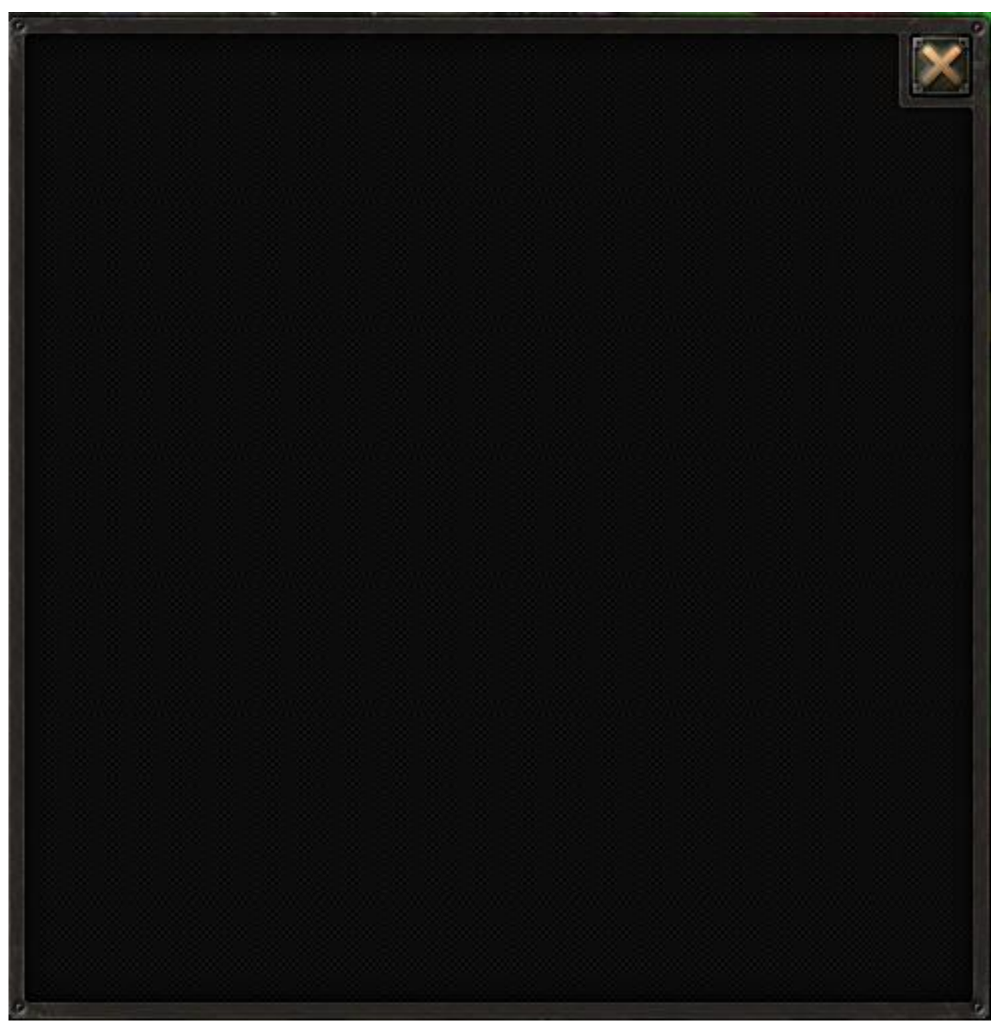

Рисунок 1.4 – Завмерле вікно асів

Кожний ас є носієм деякого рівня, умовно – від першого до третього, як бачимо на рисунку 1.3 – ас дає бонуси до трьох характеристик, рівень аса визначає розмір цього бонусу. Очевидно, що аси є об'єктами в коді гри, на це

вказує вміст файлів гри, де присутній текст схожий на код програми (див. рис. 1.5).

```
4     ### Fighters ###
5     fighter_good = {
6         type = { fighter interceptor }
7         chance = 0.9
8         effect = {
9             air_attack_factor = 0.03
10            air_maximum_speed_factor = 0.03
11            air_agility_factor = 0.05
12        }
13    }
14    fighter_unique = {
15        type = { fighter interceptor }
16        chance = 0.4
17        effect = {
18            air_attack_factor = 0.06
19            air_maximum_speed_factor = 0.05
20            air_agility_factor = 0.1
21        }
22    }
23    fighter_genius = {
24        type = { fighter interceptor }
25        chance = 0.05
26        effect = {
27            air_attack_factor = 0.1
28            air_maximum_speed_factor = 0.08
29            air_agility_factor = 0.15
30        }
31    }
```

Рисунок 1.5 – Частина файлу гри із характеристиками асів трьох рівнів

Подивившись уважно дійсно можна побачити щось схоже на С-подібну мову програмування, якою прописані три рівні асів, для кожного з яких вказані ті самі бонуси та їх значення, що вони дають в грі.

Одразу треба зазначити, що для цієї гри такі файли – нормальна практика, багато ігрових налаштувань, що встановлюють стан ігрових сутностей прописані саме таким чином – через текстові файли (ще приклад на рисунку 1.6).

```
equipments = {

    artillery_equipment = {
        year = 1934

        is_archetype = yes
        picture = archetype_artillery_equipment
        type = {
            artillery
            infantry #adding infantry type
            #motorized #Removing motorized type
        }
        group_by = archetype

        interface_category = interface_category_land

        #Misc Abilities
        reliability = 0.8

        #Defensive Abilities
        defense = 10
        breakthrough = 6
        hardness = 0
        armor_value = 0

        #Offensive Abilities
        soft_attack = 25
        hard_attack = 2
        ap_attack = 5
        air_attack = 0

        #Space taken in convoy
        lend_lease_cost = 4

        build_cost_ic = 3.5
        resources = {
            tungsten = 1
            steel = 2
        }
    }

    artillery_equipment_1 = {
        year = 1934

        archetype = artillery_equipment
        priority = 50
        visual_level = 0
```

Рисунок 1.6 – Частина файлу ігрової сутності “артилерія”

Власний досвід показав, що зміна значень в таких файлах дійсно впливає на ці значення в самій грі. Воно і не дивно, характер таких записів дуже схожий на конструктор класів в С-подібних мовах програмування, крім того, в таких файлах є коментарі, а на офіційній Wiki гри цілі статті, від розробників, що пояснюють значення записів в таких файлах та навіть містять деякі поради щодо їх змін. Таким чином, ми цілком виправдано можемо будувати своє уявлення про реальний вихідний код гри на основі таких файлів.

Отже, можна точно сказати, що клас асів має як мінімум 3 атрибути числового типу даних, в яких зберігаються значення бонусів. Крім того, було знайдено файл, що відповідає за набір імен, прізвищ та позивних, що можуть мати аси при їх генерації (див. рис. 1.7).

```
48  GER = {
49      male = {
50          names = {  Maximilian Felix Erich Gerhard Gunther Otto Walter Wilhelm Heinz Hermann Heinrich
51              Theodor Hans Anton Joachim Max Emil Peter Fritz Otto Adolf Tobias Staffan Marco Karl
52          }
53      }
54      female = {
55          names = { Jessica Emma Hanna Sofia Marie Hildegard }
56      }
57      surnames = {
58          Muller Schmidt Schneider Fischer Meyer Weber Schulz Wagner Hartmann Rall Barkhorn
59          Becker Hoffmann Graf Erhler Hafner Lipfert Brendel Stotz Kirschner Lang Sturm
60          Beisswenger Duttmann Wilbs Berglen Behrmann
61          }
62      callsigns = { "Bubi" "Assi" "The Black Devil" "The Blond Knight Of Germany" "Dolfo" "Fighter of Libau"
63      "Petit Rouge" "Pritzl" "The Red Baron" "The Red Battle-flyer" "The Red Knight" "Reeste" }
64
```

Рисунок 1.7 – Вміст файлу із можливими іменами, прізвищами та позивними асів

І дійсно, на рисунку 1.3 можна помітити, що у асів є імена, позивні та прізвища, що не дивно, крім того, в файлі імен можна помітити розділення на жіночі та чоловічі, що також є очікуваним, адже в грі дійсно є жінки аси. Таким чином можемо впевнено додавати ще мінімум 4 атрибути до класу асів.

Сумарно маємо 7 атрибутів, навіть якщо припустити, що рядкові типи даних займають по 40 байтів кожна, що залежить від довжини самого рядка, то отримаємо смішні об’єми пам’яті, що необхідні для зберігання інформації про цих асів. Ніяких функцій, що включають в себе складні математичні обчислення, аси

явно не мають, адже лиже додають до характеристик крила літаків три бонуси шляхом множення. Тут навіть немає сенсу обчислювати який об’єм пам’яті займають такі об’єкти.

То у чому ж проблема? Все доволі просто, якщо бути знайомим із нюансами розробки інтерфейсів Web проектів. Вікно асів містить у собі всіх асів, як вільних, так і прикріплених до крил, тому інформації в такому вікні доволі багато. У Web проектах в аналогічних ситуаціях інформацію для відображення відображають поступово – як тільки вона прийде на клієнт, а та, що ще не була отримана, тимчасово замінюється деякою картинкою, або анімацією, що виглядає як завантаження інформації (див. рис. 1.8).

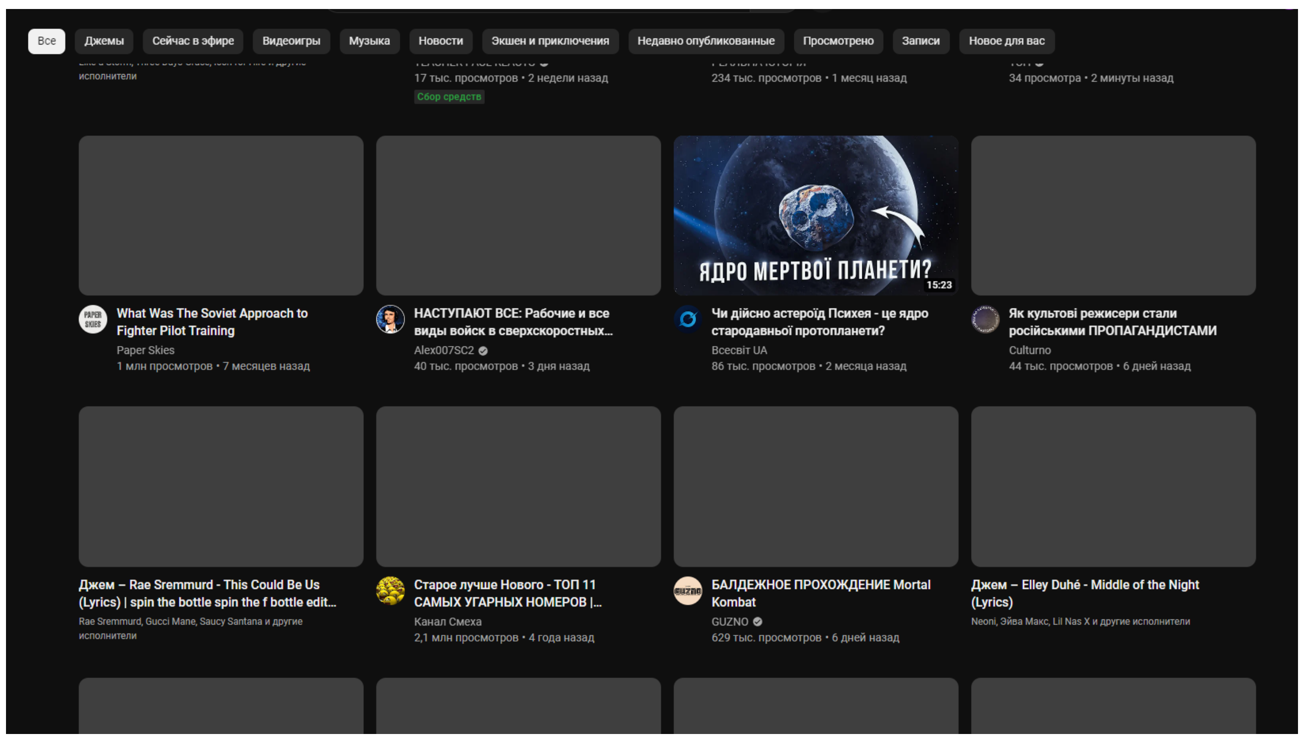

Рисунок 1.8 – Поступова прогрузка зображень обкладинок відео на YouTube

Такий підхід дуже гарно працює з точки зору юзабіліті – користувач не буде змушений чекати поки вся інформація прийде на клієнт та відобразиться повністю. Не зрозуміло – в зв’язку з чим розробники “Hearts of Iron IV” не використали подібний підхід, адже при вищевказаній проблемі можна явно

зрозуміти, що спочатку завантажується вся інформація (список асів), а потім тільки відображається.

Іншою і найбільшою проблемою гри є її уповільнення при великих кількостях юнітів, які в грі є дивізіями, які і приймають участь у боях. У кожної дивізії є її шаблон – по суті її склад (див. рис. 1.9). Від шаблону залежить те, як буде поводити себе дивізія в процесі виконання, з програмної точки зору, своїх двох основних функцій – переміщення та битви з ворожими дивізіями. Так, склад дивізії впливає на те, з якою швидкістю вона буде рухатися по тій чи іншій місцевості, що означає додаткові навантаження на систему при її вирахуванні.

Рисунок 1.9 – Конструктор дивізій, де налаштовується їх склад

Кожний батальйон потребує деякого спорядження для укомплектування, спорядження, в свою чергу має деякі, характеристики, таким чином характеристики дивізії вираховуються по суті через спорядження що вона має. Можна мати безліч шаблонів дивізій, і будь-яка дивізія його має. Дивізії можна об’єднувати в армії, зокрема для легшого менеджменту, а армії в групи армій.

На перший погляд, наче нічого не говорить про те, що така система складна для «заліза» ПК, адже очевидно, що навіть при такій не малій кількості характеристик, що можна побачити на рисунку 1.9, це лише чисельні атрибути,

що не займають багато пам’яті. Проте тут є декілька нюансів, що все ускладнюють.

По-перше, що кожна країна, що присутня в партії, окрім країни гравця, знаходиться від управлінням штучного інтелекту, згадаємо скільки їх було історично на момент 1936 року. Штучний інтелект (далі – ШІ) дуже полюбляє використовувати багато різних шаблонів дивізій. Якщо зазвичай гравець матиме від одного до максимум шести шаблонів піхоти, то ШІ дуже часто має мінімум 3 і активно їми користується. В зв’язку з тим, що характеристики кожної дивізії певного шаблону різні, результати двох основних дій – переміщення та бою – будуть відрізнятися, що дещо ускладнює розрахунки в деяких випадках. Більш того, «комп’ютер» дуже полюбляє робити занадто велику кількість дивізій, особливо гостро це відчувається на великих країнах (СРСР, Німеччина, США, Франція тоще), де їх кількість може сягати 400 одиниць.

По-друге, ШІ дуже дивно поводить себе у менеджменті своїх дивізій. Якщо гравець може мати 5-15 армій, в залежності від можливостей країни, за яку він виступає, то ШІ ділить свої дивізії на аномально велику кількість армій (див. рис. 1.10).

Рисунок 1.10 – Поділ дивізій на армії (групи) гравця(зверху) та штучного інтелекту (знизу)

Таким чином виходить, що буквально декілька дивізій із можливих 24, що є лімітом управління генерала армії, входять в кожну армію. Так, це все пов’язане з тим, що ШІ полюбляє чітко ділити відповідальність між групами дивізій і, якщо

йому треба виконати два різних завдання в одному регіоні, або два однакових в сусідніх регіонах, він буде використовувати дві окремі армії, що і призводить до такого розбиття дивізій, що можна бачити на рисунку 1.10

Більш того, ШІ активно ділить свої армії на театри бойових дій, тобто на рисунку 1.9 зображені не всі групи дивізій, а тільки ті, що знаходяться на одному театрі. На інших ситуація найчастіше аналогічна (див. рис. 1.11).

Рисунок 1.11 – Інші театри бойових дій зі своїми арміями, що встановив ШІ

По-третє, в грі присутній ще один штучний інтелект, що відповідає не за керування країною а за виконання наказів арміям, такий собі помічник, аби не керувати кожною дивізією окремо – гравець може скористатися наказами для армії. Одними із таких наказів є формування фронту (розташування дивізій на кожній клітині на кордоні з іншою країною) і наступ, вони є дуже показовими у роботі цього помічника. На рисунку 1.12 можна побачити процес наступу, в ході

якого ШІ переміщує дивізії вздовж лінії фронту (зелені стрілки) на, як йому здається, більш правильні позиції.

Рисунок 1.12 - Менеджмент дивізій штучним інтелектом під час наступу

Як можна побачити, такий автоматичний менеджмент змушує дивізії значно змінювати свою локацію вздовж лінії фронту, майже не проводячи при цьому наступальних дій (червоні стрілки), хоча такий наказ є активним. По суті це є зайвими діями, що звичайно потребують деяких розрахунків, з програмної точки зору.

Помножимо всі три вище вказані нюанси одне на одного і матимемо систему із більш ніж тисячею різних об’єктів із немалою кількістю атрибутів, що активно виконують свої функції, засновані на цих атрибутах, подекуди явно марно. На перший погляд не вражає, у системах штучного інтелекту, що ми

розглянули вище може бути і значно більше об’єктів і більш складні розрахунки. Проте, ми забули вказати ще дві дуже важливі моменти.

Перше – всі дії в грі прораховуються для кожного об’єкту кожен тік (від англійського “tick” - один цикл ігрового часу), що дорівнює одній ігровій годині. Тобто об’єкт може виконати 24 дії протягом ігрового дня. Таким чином, чим більше буде об’єктів, тим більше необхідно виконати програмних розрахунків в одному тіку, тим повільнішою буде гра, що явно неприємно для гравця.

Друге – під час бою дивізія може втрачати особовий склад та оснащення, що призводить до неповної укомплектованості до наступного поповнення (див. рис. 1.13).

Рисунок 1.13 – Параметри дивізії та статус її наповненості

Повнота особового складу та спорядження називається в грі міцністю та приймає значення від 1 до 100%. Так от всі характеристики дивізії приводяться до відсотку, що відповідає відсотку її міцності. Таким чином, для кожного об’єкту,

що є дивізією треба перераховувати значення всіх характеристик в той момент, коли вона втратила або набула міцність.

Саме такі нюанси роботи гри змушують систему ближче до середини партії працювати помітно повільніше за самий її початок. І це лише наземний бойовий аспект гри.

Заради дослідницького інтересу та повноти дослідження – приблизно прорахуємо який об’єм пам’яті займає кожна дивізія, як ігровий юніт та об’єкт та скільки залежностей та/або прорахунків треба врахувати після кожного тіку.

Пустий шаблон дивізії налічує 28 відкритих параметрів (див. рис. 1.14), 27 з яких представлені десятковими числами.

Рисунок 1.14 – Базові характеристики дивізії

Крім того, наявні ще 24 приховані параметри, вони знаходяться в полі «Модифікатори» графічного інтерфейсу (їх можна побачити на рисунках 1.9 та 1.13). Вони зумовлені наявністю 8 різних типів місцевості в грі, які накладають певні штрафи або бонуси до чотирьох характеристик дивізії під час бою на них за допомогою трьох модифікаторів (три іконки поменше під кожною іконкою типу місцевості, 8 * 3 = 24). Такі модифікатори залежать від складу дивізії, тому порожня має нульові їх значення, через що вони і не відображаються на рисунку 1.13. Всі ці параметри в грі є відсотками, тому логічно, що вони теж є десятковими числами. Будь-яка не порожня дивізія, як вже було сказано вище, буде мати додаткові параметри, що відображають кількість спорядження в дивізії, яке, як і єдиний базовий останній параметр "Людські ресурси", є числами цілими і, більш того, повинна мати максимальне значення цього параметру та поточну. Тобто маємо ще один базовий параметр та ще декілька, що залежать від шаблону дивізії, тому для експерименту візьмемо таку дивізію, як на рисунку 1.9 та 1.13, це одна й та ж і вона доволі звичайна для гри. Вона має 5 видів спорядження, що дає нам іще 10 параметрів цілочисельного типу даних, звісно – якщо шаблон дивізії буде мати додаткові відмінні батальйони, то і видів спорядження зазвичай стає більше, іноді більше ніж на один.

Таким чином, маємо 52 десяткових числа та 12 цілих, тобто, зважаючи на особливість гри, можемо сказати, що це 52 змінні типу float та 12 типу short, тобто використавши базову математику – 232 біти на одну дивізію, тобто на один об'єкт. На перший погляд це зовсім мало, навіть за умови існування більше 1000 таких об'єктів, проте, головне навантаження на комп'ютер створює кількість та складність обчислень в процесі роботи таких об'єктів. Це у свою чергу залежить від залежності параметрів одне від одного в рамках логіки гри (ланцюжок залежностей). Як вже було згадано – неповністю укомплектована спорядженням та людьми дивізія має занижені характеристики, це можна побачити на рисунку 1.15.

Рисунок 1.15 – Вплив неповної укомплектованості дивізії на її характеристики

І ось тут розмір тих самих параметрів впливає на швидкість обчислень, адже, як було вивчено протягом курсу «Архітектура комп'ютерів» та в процесі ознайомлення із роботою БЕОМ (базова електронна обчислювальна машина), чим більше бітів займає змінна, тим більше знадобиться тактів роботи машини (тобто часу), аби провести будь-яку математичну операцію над нею. Складність же обчислень підвищується із використанням банально більш складних арифметичних дій. Так, якщо дія складання двох змінних це одна умовна базова дія, то процес множення це процес додавання першого множника до самого себе кількість разів рівна другому множнику, тобто тут кількість умовних базових дій буде рівна другому множнику. Тому необхідно зрозуміти яку кількість залежностей параметри мають одне на одного в рамках одного об'єкту.

Для цього візьмемо дивізію із таким же шаблоном як і на рисунку 1.8, на рисунку 1.16 можна побачити характеристики обведені червоним кольором – базові характеристики (те саме спорядження та особовій склад), які впливають на всі інші характеристики, що обведені жовтим кольором.

Рисунок 1.16 – Групи залежних (жовта) та впливаючих (червона) характеристик

Таким чином, маємо вплив 6 параметрів на 22 і, що дуже важливо, дійсно кожен червоній параметр впливає на кожен жовтий. Таким чином маємо 6 * 22 = 132 залежності всі з яких мультиплікативні. І це лише при умові, що дивізія може мати в собі різні моделі одного й того ж оснащення, що будуть мати трохи інші характеристики, що в результаті може збільшити кількість обчислень як на декілька одиниць, так і в декілька разів (друге трапляється набагато частіше) Згадуємо, що при множенні двох чисел одного розміру треба в два рази більше оперативної пам’яті лише для зберігання результату. Потім згадуємо, що таких об’єктів може бути більше 1500 одночасно в рамках однієї партії гри и вже отримуємо 198 000 обчислень, що треба провести за один тік гри і це лише

перерахування характеристик об’єктів після їх участі у бою. Тобто ми опускаємо самі прорахунки бою, небойові втрати, повністю опускаємо повітряну складову гри – авіацію, яка теж є об’єктами із своїми характеристиками, задачами і, як наслідок, обчисленнями, також опускаємо все, що стосується економічної складової гри, де процеси розрахунків відбуваються постійно, а ще опускаємо усілякі події, що мають строк, умови виконання, або невиконання тощо.

Таким чином, маємо вражаючу кількість обчислень, що треба провести за один тік гри, який відповідає одній ігровій годині, тобто – поки всі прорахунки в рамках одного тіку не закінчяться – гра для гравця не буде рухатися, приймаючи до уваги тематику гри (Друга світова війна), зрозуміло, шо одина ігрова година це дуже невеликий проміжок часу, за який нічого суттєвого в грі не станеться, тобто велика кількість об’єктів прямо впливає на швидкість ігрового процесу і комфорт гри.

Дуже схожою на “Heart of Iron” є гра “Stellaris” від того ж виробника. Гра має космічний сетінг і головною ігровою сутністю, що нас цікавить є одиниця населення (англійською має назву *pop* від *population*, тому далі будемо скорочувати до “поп”). Попи розміщуються на планетах, що теж є ігровими сутностями, а не скупченням населення. Поп є ключовою фігурою економіки в грі, адже саме він займає робоче місце, що надає планета, та виробляє на робочому місці ресурси.

Перед обговоренням складності системи характеристик ігрових сутностей, треба описати ланцюжок залежностей та впливів характеристик одних сутностей на інших:

– планета, як небесне тіло, впливає на деякі характеристики попів (наприклад, на рівень пристосованості до клімату планети), та на продуктивність роботи попів;
– ці деякі характеристики окремих попів (які ще на додачу залежать від політичного строю імперії) впливають на характеристики планети як скупчення населення (наприклад, рівень стабільності);

– в свою чергу ці характеристики впливають вже на фінальну і найважливішу характеристику – рівень продуктивності;

Крім того, на рівень продуктивності впливають деякі характеристики кожного окремого попу та модифікатори імперії та планети.

Тепер повернемося до системи характеристик попів як ключового об’єкту системи економіки і, разом з тим, всіх обчислень, які занесені в програмний код та проводяться комп’ютером під час гри. По-перше, кожний поп має вид (расу), кожна імперія (яких може бути до 36) є імперією одного виду (іноді більше, але будемо брати мінімальні значення), кожен вид має ті самі характеристики (в грі називаються ознаками), які разом є шаблоном виду, в процесі гри вид може втрачати або набувати ознаки (не більше 6 загальних ознак одночасно, плюс можуть буди унікальні зумовлені самою сутністю раси та завжди одна ознака вподобаного клімату), тобто змінювати шаблон, і попи однієї раси можуть мати різні шаблони (див. рис. 1.17).

Рисунок 1.17 – Панель шаблонів однієї раси

По-друге, на кожній планеті всі типи робочих місць поділяються на три рівні – робочий, спеціаліст, та правитель, один тип може знаходитися лише на

одному рівні. Займаючи певне робоче місце поп отримує той самий рівень, що по суті є розподілом на такі собі соціальні шари.

По-третє, механікою політичного режиму в імперії зумовлена наявність такої ігрової сутності як політична фракція, зазвичай їх три в кожній імперії і кожен поп є прихильним якої-небуть фракції.

Знаючи, що в рамках двох соціальних шарів існує більше 10 різних робочих місць (в одному з яких 4 є найпоширенішими, а у другому 7), а у третьому 7 (2 найпоширеніших), ми можемо взяти середні показники кількості вказаних вище особливостей та порахувати кількість унікальних ігрових сутностей (об’єктів), що одночасно приймають участь у процесі гри та навантажують комп’ютер. Так нехай маємо в рамках кожної імперії 1 вид із 2 шаблонами, 3 фракції, 3 соціальні шари, в середньому по 4, 7 та 2 різні роботи:

$$2 * 3 * (4 + 7 + 2) = 78$$

Тобто, 78 унікальних екземплярів класів в кожній імперії, не враховуючи унікальність планет в рамках імперії, що робить місцевих попів відмінними від аналогічних на інших планетах. Враховуючи максимум в 31 активну імперію маємо більше 2400 різних унікальних об’єктів в рамках однієї партії. Можливість знаходження кількох різних рас в рамках однієї імперії і однакових рас в рамках різних імперій можемо впевнено множити це значення на число від 2 до 31^2 (в залежності від поширення стартових раз між усіма існуючими імперіями в партії) і ще не забуваємо помножити на кількість планет в такій грі, що може становити більше 500, адже кожна планета, як уже було сказано впливає на характеристики попів, що на ній проживають. Навіть якщо брати більш-менш приземлені числа, то маємо не один мільйон унікальних об’єктів.

Виглядає дуже жахливо, але на практиці такої різноманітності не буває, адже сама гра обмежує деякі комбінації в результаті оптимізації економіки (поп із бонусом до роботи фермером в пріоритеті буде фермером, а не науковцем) і деякі числа ми брали в найгірших разі найгірших ситуацій, та й існуючих в конкретній

партії різноманітностей попів не може бути більше за саму їх кількість в партії. Повернемося до числа 78 - на справді на стільки великим воно бути не може, через оптимізацію економіки самим кодом гри (як було сказано вище) та через банальну неможливість деяких комбінацій. Тому всі подальші обчислення дадуть набагато менші результати, які насправді будуть в межах від 1000 до 2000 унікальних об’єктів.

Виглядає не дуже вражаючим, особливо згадуючи, що ознак в одному шаблоні раси (по суті кількість полів в програмному об’єкті) в основному не більше 6, але згадуємо ланцюжок залежностей та впливу показників одне на одного, розуміємо множимо всі зі залежності на кількість унікальних попів на планеті і отримуємо цілу купу обчислень, що необхідно провести, особливо, якщо щось змінюється на самому початку ланцюжка. Для наглядної демонстрації на рисунку 1.18 можна побачити скільки модифікаторів до рівня продуктивності (останнього елемента ланцюга) застосовується до дослідника.

Рисунок 1.18 – Модифікатори до рівня продуктивності

Складність обчислень в вказаному вище ланцюжку та в процесі автоматичної оптимізації економіки кодом гри дуже яскраво описує розмір файлу

гри із параметрами наявних типів роботи для спеціалістів, що для всіх цих обчислень використовуються -2301 рядок (див. рис. 1.19).

```
01_ruler_jobs.txt    02_specialist_jobs.txt ×    03_worker_jobs.txt
F: > Games > Stellaris > common > pop_jobs > 02_specialist_jobs.txt
2289
2290        weight = {
2291            weight = @high_prio_specialist_job_weight
2292            factor = 1.2
2293            add = 5 #Somehow pops preferred to work as squires than as knights. No more.
2294            modifier = {
2295                mult = value:job_weights_modifier|JOB|bureaucrat|RESOURCE|unity|
2296            }
2297            modifier = {
2298                mult = value:job_weights_research_modifier|JOB|researcher|
2299            }
2300        }
2301    }
2302
```

Рисунок 1.19 – Кінець файлу гри із програмною реалізацією об’єктів

Враховуючи, що попи та їх робота це найбільша частина всіх об’єктів та процесів, що проходять в грі, немає ніякого сумніву, що цей аспект гри впливає на її оптимізацію найбільше, да що та говорити, коли самі розробники в меню налаштувань нової партії самі застерігають не зловживати підвищенням швидкості росту попів в грі (див. рис. 1.20).

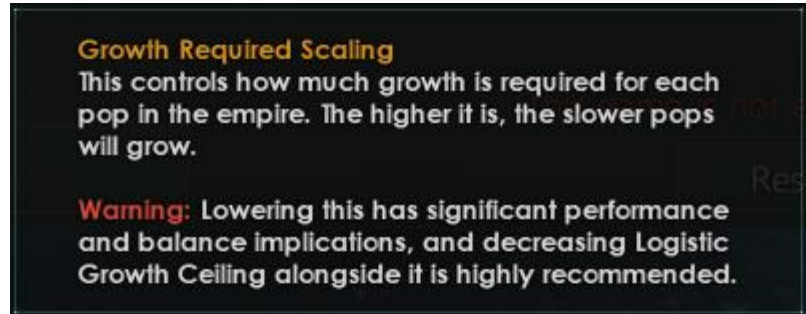

Рисунок 1.20 - Застереження розробників щодо продуктивності при більших кількостях попів (об’єктів) у грі

Наступна гра на розгляд – “Warcraft 3”, що вийшла в далекому 2003му році. Це стратегія в реальному часі, де найпоширенішим є режим гри, в якому кінцевою

метою гравця є знищення «бази» супротивника. За час її існування уже, мабуть, усі, хто мав з нею справи, знає її проблеми в оптимізації. Всі вони тягнуться від однієї особливості гри – вона працює в режимі навантаження лише одного ядра центрального процесора. То і не є дивно, в далекому 2003му році ані розробники, ані користувачі і не могли уявити собі процесори із 16, 32 та більше ядрами.

Найбільш яскравою проблемою, що витікає звідси, стала проблема завмирання юнітів, яка проявлялася частіше за все в процесі їх переміщення. На рисунку 2.21 можна побачити запит одного із користувачів на форумі із цією проблемою, з якою від зіштовхнувся під час розробки свого контенту до гри, та порада іншого користувача з вирішення цієї проблеми [5].

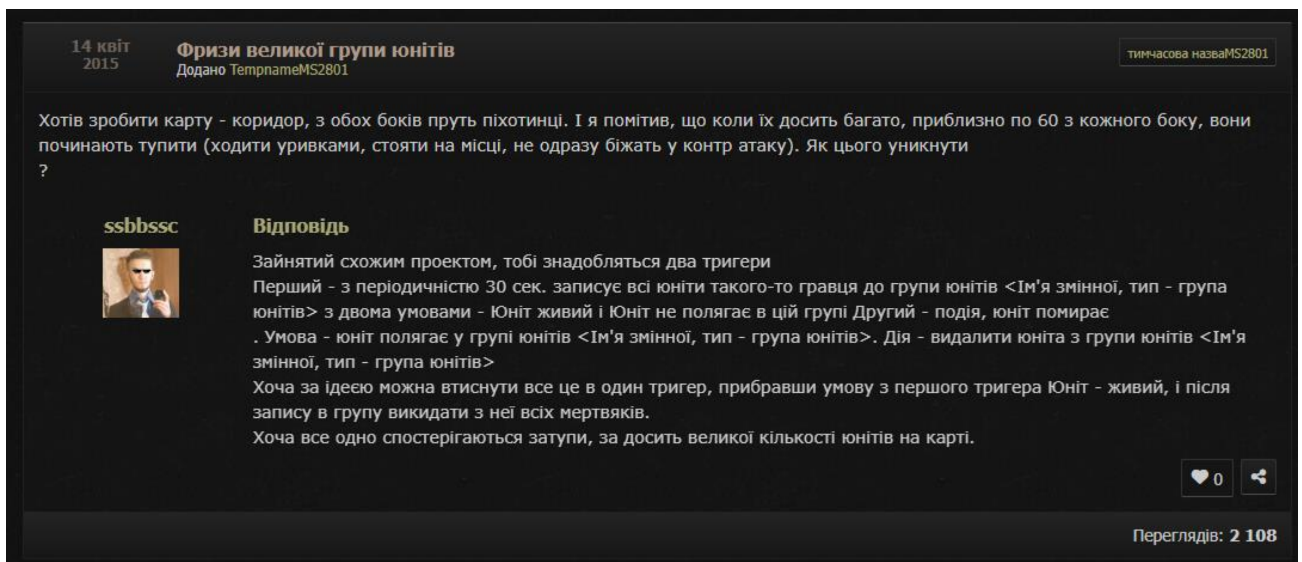

Рисунок 2.21 – Опис проблеми завмирання юнітів від одного із користувачів та варіант її рішення

Також причиною таких проблем став трохи дивний дизайн гри. Гравець може обрати юнітів групою максимум лише з 12 одниць (див. рис. 2.22).

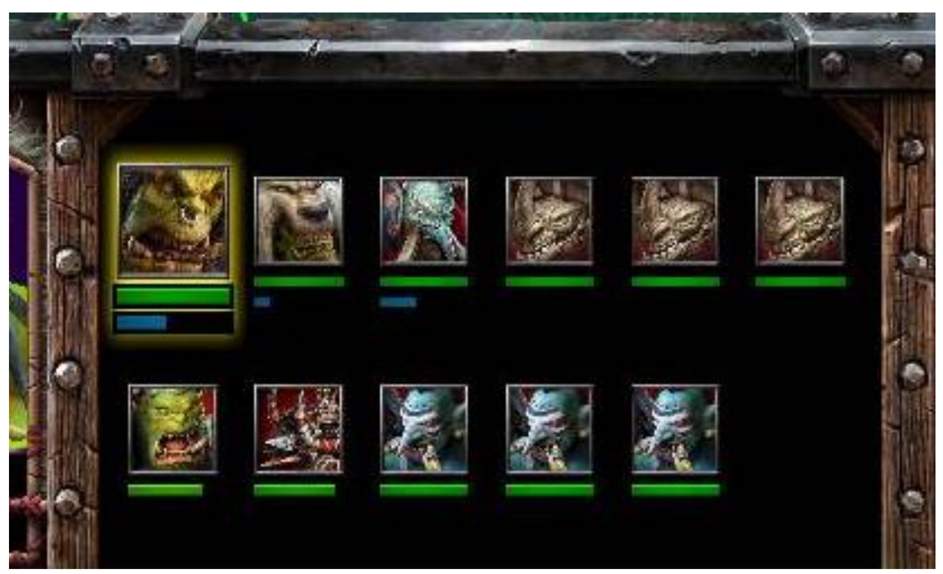

Рисунок 2.22 – Вигляд обраної групи юнітів

Звичайно це пов'язане із вихідним кодом гри і його механіками, тому навіть штучний інтелект гри віддає накази не одразу всім необхідним юнітам, а ось таким їх групам. Сюди ж додається проблема перевизначення груп. Справа у тому, що якщо юніт загинув, то ви його звісно в групі не побачите, але на програмному рівні він там ще є і він наче приймає участь у грі і приймає ті ж команди, що і вся група. В разі гравця, така проблема менш критична, адже йому просто дуже важко керувати юнітами не використовуючи групи, тому при їх втратах він буде поповнювати групу новими юнітами, таким чином фактично їх перевизначаючи. А от із штучним інтелектом справа інша, якщо розробник, який писав конкретну поведінку конкретному «комп'ютерові», не врахував таку особливість, то ШІ може нехтувати перевизначеннями груп, що неминуче призведе до такої кількості «фантомних» юнітів, що робота гри буде некомфортною для користувача.

Крім того, цю проблему можна пов'язати із механікою вишикування юнітів. При її активації юніти будуть намагатися тримати стрій під час переміщення таким чином, аби умовні стрільці були за спинами своїх умовних лицарів, а чаклуни ще більш позаду, що фактично ділить кожну групу із такими юнітами на ще 3 підгурпи.

2. ВИКОРИСТАННЯ ОТРИМАНИХ ЗНАНЬ НА ПРАКТИЦІ

2.1. Проведення експерименту

Протягом своєї роботи в компанії із розробки ігор я мав справу із програмним кодом частини гри, який відповідає саме за обчислення, які дуже схожі на все те, із чим ми познайомилися вище. З метою нерозголошення конфіденційної інформації підприємства сама суть гри буде опущена, а увага сконцентрована на технічній частині програми, що цікавить нас саме в контексті даної роботи, всі демонстраційні матеріали будуть показувати лише необхідну для експерименту інформацію.

Почнемо з того, що як такої великої кількості об’єктів там немає, проте, там присутня велика кількість однакових математичних дій, над постійно різними вхідними даними. Згадуючи як працюють проаналізовані вище ігри з їх системою тіків, можна сказати, що с технічної точки зору це абсолютно задовільні для порівняння процеси, адже не важливо, що об’єктів багато – поки не виконаються всі дії в рамках одного тіку – стан гри не зміниться, а тих рамках можна сказати, що дії виконуються поступово, як і в нашій програмі для експерименту. Найважливіше для нас – розуміти, що кількість доволі базових дій в обох іграх порівнянна.

Так як програма вже готова, використовується вона далеко не вперше і вже відслужила деякий час, очевидно, що над нею достатньо працювали аби вона відпрацьовувала правильно та не мала великих і явних недоліків в оптимізації. Тому в ході експерименту ми спробуємо навпаки – погіршити оптимізацію, аби зрозуміти яка могла бути різниця, якби над нею не попрацювали належнім чином.

Сама гра приймає на вхід доволі прості масиви даних, ці дані за самою суттю гри мають бути натуральними числам, найчастіше не більше 10, тому для них використовується тип даних sbyte, із метою не займати зайвої оперативної пам’яті та прискорити обчислення (див. рис 2.1 та 2.2).

```
= new sbyte[][]
{
    //there was just an inportant comment, I'd better remove it
new sbyte[]{1,1,1,3,3,3,2,2,5,5,6,6,6,4,4,2,2,2,4,4,0,7,3,3,3,1,1,1,2,2,2,2,0,1,1,1,0,3,3,3,0,5,5,5,1,1,1,0,2,2,2,3
new sbyte[]{6,6,6,7,0,1,1,1,3,3,3,2,2,2,6,6,6,6,3,3,3,7,0,2,2,2,0,4,4,4,3,3,3,1,1,1,5,5,5,0,2,2,2,6,6,6,4,4,4,4,2,2
new sbyte[]{6,6,6,0,5,5,5,5,0,4,4,4,4,6,6,6,0,4,4,4,5,5,5,3,3,3,7,2,2,2,4,4,4,6,6,6,7,3,3,3,3,5,5,6,6,4,4,4,4,2,2,2
new sbyte[]{5,5,5,5,1,1,6,6,6,4,4,4,4,6,6,6,6,0,4,4,4,5,5,5,3,3,3,7,6,6,6,2,2,2,4,4,4,5,5,5,0,6,6,6,4,4,4,3,3,7,6,6
new sbyte[]{7,5,5,5,3,3,3,4,4,5,5,5,5,2,2,6,6,0,4,4,4,5,5,5,7,6,6,4,4,4,3,3,3,6,6,4,4,4,4,1,1,1,5,5,5,0,6,6,6,7,1,1


    //there was too
/*new sbyte[]{1,1,1,3,3,3,2,2,6,6,6,4,4,2,2,2,4,4,0,7,3,3,3,1,1,2,2,2,0,1,1,1,0,3,3,3,0,5,5,5,1,1,1,0,2,2,2,5,5,5,5
new sbyte[]{6,6,7,0,1,1,1,3,3,3,2,2,2,6,6,6,6,3,3,3,7,0,2,2,2,0,4,4,4,3,3,3,1,1,1,1,3,3,3,5,5,5,0,2,2,6,6,6,4,4,4,0
new sbyte[]{6,6,6,0,5,5,5,5,0,4,4,4,6,6,6,0,4,4,4,3,3,3,3,1,1,5,5,7,2,2,4,4,4,6,6,7,3,3,3,5,5,6,6,4,4,4,2,2,2,5,5,5
new sbyte[]{6,6,5,5,5,1,1,6,6,6,4,4,4,6,6,6,0,4,4,4,5,5,5,3,3,3,7,6,6,6,2,2,2,4,4,4,5,5,5,0,6,6,6,4,4,4,4,5,5,5,3,3
new sbyte[]{7,5,5,5,3,3,4,4,5,5,5,2,2,2,6,6,6,0,4,4,5,5,5,7,6,6,6,4,4,4,3,3,3,4,4,4,1,1,1,5,5,5,0,6,6,6,6,7,1,1,1,4
```

Рисунок 2.1 – Частина вхідних даних програми

```
lines = new sbyte[][]
{
new sbyte[] { 1, 1, 1, 1, 1 }, new sbyte[] { 2, 2, 2, 2, 2 }, new sbyte[] { 0, 0, 0, 0, 0 }, new sbyte[] { 1, 2
new sbyte[] { 0, 1, 1, 1, 0 }, new sbyte[] { 0, 1, 2, 1, 0 }, new sbyte[] { 2, 1, 0, 1, 2 }, new sbyte[] { 0, 0
};
```

Рисунок 2.2 – Частина вхідних даних програми

Крім того, наявні ще декілька параметрів, що мають тип даних sbyte та є простими змінними значенням в одне число і ще два параметри такого ж формату як і вказані вище.

Ідея експерименту в тому, щоб змінити тип даних цих змінних на int, що в 4 рази більший за обсягом пам’яті ніж sbyte, та виміряти час виконання програми в обох випадках – до зміни і після, адже в процесі цієї роботи ми зрозуміли, що розмір типів даних, що використовують змінні, впливає на швидкість не тільки обчислень а й процесів порівняння змінних [4]. Насправді саме ці дані не приймають участі в процесах множення або сумування, а лише порівняння, тому різниця має бути не дуже великою, але одночасно з цим і сама програма потребує великої кількості повторень, які можна задати безпосередньо в ній, аби результат був коректним. Отже змінимо типи даних та запустимо програми на 500 мільйонів повторень кожна, результати до та після можна побачити на рисунку 2.3

```
Microsoft Visual Studio Debug Console

Game :
Spins : 500000000
257.177 s.
Total     : 92.10% +- 0.09%%   Total Max     : 9375.5
```

```
Microsoft Visual Studio Debug Console

Game :
Spins : 500000000
261.992 s.
Total     : 91.79% +- 0.09%%   Total Max     : 9906.5
```

Рисунок 2.3 – Результати оригіналу (зверху) та після деоптимізації (знизу)

Як бачимо – різниця в результатах складає лише приблизно 5 секунд і у відсотках це приблизно 2%, що й не дивно, адже, як вже було сказано – ті змінні, яких ми торкнулися, не приймають участі саме в обчисленнях.

для такої незначної зміни виглядає насправді доволі непогано, враховуючи той факт, що в програмі відсутня ланцюжкова залежність параметрів одне від одного, або залежності одного параметру від декількох інших.

Тепер спробуємо внести зміни в ті дані, що безпосередньо приймають участь у розрахунках. В рамках програми є одне значення, що вираховується процесом як суми так і множення деяких змінних. Спробуємо перевести тип даних у змінної цього значення та інших змінних, від яких вона залежить, із int у доволі короткий тип ushort (програма працює на C#), він у 2 рази коротший (2 біти) і ця змінна не може бути від’ємною, тому беремо саме ushort. Аби точно не перевищити допустиме значення, що задаю тип даних, використаємо іншу гру, де це визначено математичною моделлю. Таким чином ми маємо отримати трохи швидше виконання програми. На рисунку 2.4 маємо результат.

```
Microsoft Visual Studio Debug Console

Game :
Spins : 500000000
275.368 s.
Total     : 92.05% +- 0.05%%  Total Max     : 710

Microsoft Visual Studio Debug Console

Game :
Spins : 500000000
267.721 s.
Total     : 92.02% +- 0.05%%  Total Max     : 680
```

Рисунок 2.4 - Результати оригіналу (зверху) та після оптимізації (знизу)

Ось тут все різниця більш відчутна – маємо різницю в 7.5 секунд, або майже 3 відсотки. Знову ж таки – мі лише змінили типи даних у змінних, що приймають участь у обчисленнях, очевидно, що більш серйозні методи оптимізації дадуть кращий результат.

3. ПОШУК МОЖЛИВИХ РІШЕНЬ

3.1. Встановлення оптимізаційної задачі

Провівши деякі дослідження в предметній області, було виявлено декілька проблем в існуючих проектах, та основі яких можна виділити наступні методи їх оптимізації:

- оптимізація графіки – графіка теж пов’язана із програмними об’єктами, оптимізація відображення об’єктів на ігровій сцені – поширена практика в сучасних іграх;
- оптимізація алгоритмів – сюди можна віднести багато різних рішень, вони сильно залежать від конкретної ситуації, це може бути оптимізація взаємодії об’єктів, переосмислення програмної реалізації їх функцій тощо;
- зміна рушія – зміна мови програмування на рівні з оновленням версії рушію може прямо вплинути на можливості розробки та оптимальної роботи кінцевого продукту;
- додавання паралелізму – паралелізм має декілька рівнів, в історії ігрової індустрії є приклади поганого використання паралелізму верхніх рівнів [5];
- спрощення/удосконалення ігрових механік – в іграх з легкістю можуть бути дуже складні механіки, що потребують багато обчислень, а іноді навіть мають «складні» графічні ефекти. Спрощення, оптимізація або навіть видалення таких механік вирішує одразу дві ці проблеми.

Введемо деякі критерії, які описують якість представлених способів оптимізації:

- збільшення FPS (кадрів в секунду) – в ході аналізу предметної області стало зрозуміло, що існуюче різноманіття ігор обґрунтовує можливість збільшення FPS, після проведення оптимізаційних заходів, у випадково обраній грі. Така характеристика дуже неоднозначна з точки зору прогнозування (не зрозуміло передчасно на скільки буде покращення

перед самим введенням оптимізації), але дуже об’єктивна в оцінці результату – більше FPS – краще. Будемо вимірювати приріст у відсотках;

- легкість реалізації – чим істотніші зміни, тим довше вони будуть впроваджуватися. Які б труднощі не виникали у процесі, все залишається питанням часу, його будемо вимірювати в місяцях. Очевидно, що чим більший час на реалізацію зміни – тим гірше, тому в подальших обчисленнях будемо використовувати це значення із знаком мінус;
- легкість подальшої підтримки – деякі рішення з оптимізації із часом можуть привести розробників у глухий кут – обраний рушій може перестати оновлюватися, більша кількість механік ускладнить подальшу розробку продукту тощо. Будемо виражати цю відносну величину у відсотках, як частина всього проекту (вихідний код, графіка, механіки), що в майбутньому може бути достатньо просто масштабуватися та змінюватися;
- витрати – будь-які зміни в уже готовому коді – додаткова робота, яка тягне за собою витрати [6]. Зміни за своєю суттю є не розробкою гри, а її підтримкою, що обходиться приблизно в 15-20% від ціни розробки, тобто приблизно в 5 разів менше, конкретні значення будемо брати на основі вартості розробки відомих ігор в тисячах доларів США. Для подальшої правильності обчислень будемо виражати цю змінну як від’ємне значення;
- вплив на геймплей – вищепредставлені зміни тим або іншим способом будуть впливати на геймплей – починаючи більш приємною «картинкою», закінчуючи прямим впливом на механіки. Цей критерій неможливо кількісно і однозначно визначити, але в середньому він може бути або кращім, після введення змін, або гіршим. Очевидно, що визначити чи є плив позитивним, або негативним, необхідно проводити опитування користувачів. Тому, під час обчислень будемо

використовувати приблизну кількість гравців у відсотках, яка вважає дану зміну більш позитивною. Забігаючи наперед, саме така шкала вимірювання буде зручною для подальших розрахунків, а ті значення, що неможливо однозначно обґрунтувати, будемо намагатися брати якнайкритичніше.

Таким чином, маємо наступну постановку задачі векторної оптимізації:

$$extremum\ Q(x), x \in D$$

Де:

- x = {Оптимізація графіки, Оптимізація алгоритмів, Зміна рушія, Додавання палелізму, Удосконалення ігрових механік} – множина альтернатив;
- $D = \{K(x_i)\}$ – критеріальний опис альтернатив

$$K(x_1) = K(\text{Оптимізація графіки}) = (20\%;\ -6\text{м};\ 90\%; -300\ \text{тис.}\$; 75\%)$$
$$K(x_2) = K(\text{Оптимізація алгоритмів}) = (10\%; -2\text{м}; 90\%; -50\ \text{тис.}\$; 85\%)$$
$$K(x_3) = K(\text{Зміна рушія}) = (0\%;\ -18\text{м};\ 98\%; -1\ 350\ \text{тис.}\$; 50\%)$$
$$K(x_4) = K(\text{Додавання паралелізму}) = (5\%;\ -2\text{м};\ 95\%; -20\ \text{тис.}\$; 75\%)$$
$$K(x_5) = K(\text{Удосконалення ігр. механік}) = (5\%; -3\text{м}; 80\%; -45\ \text{тис.}\$; 90\%)$$

Треба зазначити, що наведені показники є приблизними значеннями та засновані на аналізі власного досвіду, досвіду колег та аналізу відкритої інформації про прецеденти в індустрії.

1.1.Виявлення рішення

Після встановлення параметрів та їх значень - побудуємо таблицю (див. рис. 3.1).

	Збільшення FPS(%)	Легкість реалізації	Легкість подальшої підтримки(%)	Витрати (тис.$)	Вплив на геймплей(%)
Оптимізація графіки	15	-9	90	-300	75
Оптимізація алгоритмів	10	-2	90	-50	85
Зміна рушія	0	-18	98	-1 350	50
Додавання паралелізму	5	-2	95	-20	75
Удосконалення ігрових механік	5	-3	80	-40	90

Рисунок 3.1 – Дані для оцінки альтернатив

Надалі будемо використовувати позначення x_i і K_j для альтернатив та критеріїв відповідно (див. рис.3.2).

	K1	K2	K3	K4	K5
X1	20	-9	90	-300	75
X2	10	-2	90	-50	85
X3	0	-18	98	-1 350	50
X4	5	-2	95	-20	75
X5	5	-3	80	-40	90

Рисунок 3.2 – Дані в математичному вигляді

Як бачимо – жодна з альтернатив не є неоптимальною за Паретто.

Маємо три величини, що вимірюються у відсотках, тільки дві з яких будуть приймати значення від 0 до 100. Дві величини – легкість реалізації і витрати – зовсім не мають граничних значень, тому необхідно провести нормування критеріїв, в нашому випадку із урахуванням min і max значень. Маємо формулу зображену на рисунку 3.3.

$$f = \frac{f_{измер} - f_{min}}{f_{max} - f_{min}}$$

Рисунок 3.3 – Формула для нормування показників

Звідси маємо таблицю пронормованих значень (див. рис. 3.4).

	K1	K2	K3	K4	K5
x1	1,00	0,56	0,56	0,79	0,63
x2	0,50	1,00	0,56	0,98	0,88
x3	0,00	0,00	1,00	0,00	0,00
x4	0,25	1,00	0,83	1,00	0,63
x5	0,25	0,94	0,00	0,98	1,00

Рисунок 3.4 – Пронормовані значення

Так як у нас всюди буде значення 0 ми не можемо використовувати мультиплікативну згортку, тому використаємо адитивну згортку. Вагові коефіцієнти це дуже тонка річ, її варто використовувати тільки тоді, коли ми розуміємо з яким типом проекту маємо справу, адже оптимізація графіка в іграх із відкритим світом об'єктивно важливіша за таку в стратегіях. Тому використаємо нормуючі множники за формулою що можна побачити на рисунку 3.5.

$$\alpha j = \frac{1}{\sum_{i=1}^{m} a_{ij}}$$ – нормуючі множники.

Рисунок 3.5 – Формула вагових коефіцієнтів для кожного критерію

На рисунку 3.5 маємо результати.

	K1	K2	K3	K4	K5	Result	D(Max - R)
x1	1,00	0,56	0,56	0,79	0,63	1,260	0,005
x2	0,50	1,00	0,56	0,98	0,88	1,265	0,000
x3	0,00	0,00	1,00	0,00	0,00	0,340	0,925
x4	0,25	1,00	0,83	1,00	0,63	1,160	0,105
x5	0,25	0,94	0,00	0,98	1,00	0,975	0,290
α	0,5	0,29	0,3	0,267	0,3		

Рисунок 3.5 – Результати

Перед описом результатів слід зазначити, що немає сенсу обирати найкращій метод оптимізації, адже нічого не заважає використовувати одразу декілька із них. Тому особливу увагу треба приділити різниці між результатами альтернатив і визначити яким альтернативам слід надати більше уваги.

1.2. Аналіз отриманого результату

Як бачимо – найбільший показник у x_2 (оптимізація алгоритмів). Так як ми брали середні показники критеріїв для всіх жанрів ігор – такий результат не є дуже дивним, адже в середньому оптимальні алгоритми будуть сильно впливати на ігри з великою потребою в обчисленнях та дуже помірно на ігри з великою графічною базою. Взагалі – значення x_1 (оптимізація графіки) майже таке саме, це і не дивно, адже в іграх, де потрібні обчислення, така оптимізація великої різниці не дасть, а от в інших навпаки.

Слід звернути увагу на останній стовбець – різниця значення x_i та максимального значення. У значення x_4 (додавання паралелізму) відставання не дуже велике. Це пов’язане з тим, що паралелізм в тій або іншій мірі можна ввести в будь-яку гру, а його непоганий вплив на роботу гри дає такий результат.

Значення x_3 (зміна рушія) виглядає дуже поганим і це не є дивним – зміна двигуна гри є дуже великою за об’ємом роботою, що скоріш направлена не на оптимізацію, а на подальшу розробку продукту, така зміна має вводитися коли результат інших змін буде вже незначним, або продукт зайде в значній мірі глухий кут з точки зору розробки програмного коду.

Ну і x_5 (удосконалення ігрових механік) вийшло доволі не малим, але різниця від найоптимальнішого результату найбільша (не враховуючи x_3), тому такий варіант найменш впливовий. І дійсно, удосконалення ігрових механік є скоріш «костилем», адже дизайн гри не має впливати на її оптимізацію, хоча дійсно – немає механіки – нема проблеми з її оптимізацією.

4. ПРОБЛЕМА ПОПЕРЕДНЬОЇ РОБОТИ

4.1. Аналіз проблеми

На початку роботи було згадано, що вибір теми був викликаний, в першу чергу, із натраплянням на проблему оптимізації системи, що була розроблена в результаті попередньої кваліфікаційної роботи. Так, при наявності одного мільйона об'єктів навантаження на оперативну пам'ять була набагато більшою, ніж математично вирахуваний об'єм, а час відпрацювання алгоритму досягав майже десяти хвилин (див. рис. 3.6).

важко прорахувати, що загальна вага одного екземпляру класу – 20 байт. При проведенні тесті з участю сумарно одного мільйону юнітів має використовуватися 20 мільйонів байт, тобто лише приблизно 19МБ, проте, пам'ять процесу займала неймовірних майже 3ГБ (див. рис. 5.1).

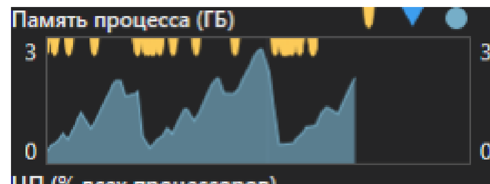

Рисунок 5.1 – Кількість зайнятої пам'яті при одному мільйоні сутностей

Крім того, було виявлено, що при таких кількостях юнітів, система дуже довго працює (див рис. 5.2)

Рисунок 5.2 – Час роботи алгоритму при одному мільйоні сутностей

Рисунок 4.1 – Опис проблеми в попередній роботі

Проаналізувавши предметну область можна чітко сказати, що при правильній побудові алгоритму програмного забезпечення, такої проблеми

виникнути не повинно. А виникла вона через не використання того ж самого методу, що ми виявили в грі “Stellaris”, а саме – групування однакових об’єктів з точки зору алгоритму. Дійсно, із програмної точки зору не має сенсу тримати в оперативній пам’яті абсолютно однакові об’єкти, якщо над над ними не буде одночасно проводитися різні дії найближчим часом. Але тут і постає та сама проблема, я кою ми зіштовхнулися вище, - іноді правила самої гри (її дизайн) змушує робити неоптимальні рішення в процесі програмної їх реалізації. Звичайно, в системі що була запропонована це не було таким випадком, проте в реальних проектах все набагато складніше, що показав власний досвід роботи в індустрії розробки ігор.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи студентом була досліджена та проаналізована тема проблем оптимізації систем з великою кількістю об’єктів в Game Dev індустрії. Були зроблені висновки на основі аналізу, які в подальшому застосувалися для проведення експерименту та формалізації фінальних висновків.

В процесі експерименту було застосовано реально існуючу програму з метою отримати якісні зміни після її оптимізації та деоптимізації.

На фінальному етапі роботи було сформовано та вирішено оптимізаційну задачу на основі отриманих попередньо даних та виявлених методах оптимізації. В результаті вирішення задачі та аналізу результату були обґрунтовані обрані методи оптимізації ігор із великою кількістю сутностей, що на програмному рівні є об’єктами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Програмування на C, C# та Java. Типи даних. URL: https://vscode.ru/articles/data-types.html (дата звернення: 10.05.2023)

2. Програмування на C, C # і JAVA. Типи даних. URL: https://jak.koshachek.com/articles/tipi-danih-programuvannja-na-c-c-i-java.html (дата звернення: 10.05.2023)

3. Wiki loginom. Еволюційний алгоритм. URL: https://wiki.loginom.ru/articles/evolution-algorithm.html (дата звернення: 15.12.2022)

4. І.М. Дудзяний, В.В. Черняхівський. Програмування мовою ассемблера. Навчальний посібник. [Електронний ресурс]: Львівський національний університет імені Івана Франка. Львів 2002. URL: https://shron1.chtyvo.org.ua/Dudzianyy_Ihor/Prohramuvannia_movoiu_asemblera.pdf?PHPSESSID=ue0vpsn2irqf6gefuopi65qc43

5. XGN Платформа для проектів з розробки та модингу ігор. URL: https://xgm.guru/p/wc3/147608

6. Michael Georgiou’s, Co-Founder of Imaginovation, article. URL: https://imaginovation.net/blog/importance-mobile-app-maintenance-cost/#:~:text=A%20ballpark%20average%20that%20an,the%20cost%20of%20maintaining%20software.towardsdatascience.

7. Pronina, D., Kyrychenko, I. "Comparison of Redux and React Hooks Methods in Terms of Performance. CEUR Workshop Proceedings, 2022, 3171, pp. 791–800.

8. Mazurova, O., Samantsov, O., Topchii, O., Shirokopetleva, M. O. Mazurova, O. Samantsov, O. Topchii and M. Shirokopetleva, A Study of Optimization Models for Creation of Artificial Intelligence for the Computer Game in the Tower Defense Genre, 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T), 2020, pp. 491-496, doi: 10.1109/PICST51311.2020.9468057.

9. Smelyakov, K., Honchar, Y., Bohomolov, O., Chupryna, A. "Machine Learning Models Efficiency Analysis for Image Classification Problem. CEUR Workshop Proceedings, 2022, 3171, pp. 942–959.