Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка емулятору ігрової платформи NES засобами мови програмування C#

Виконав: студент IV курсу, групи СП-42
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

(підпис) (прізвище та ініціали)

Керівник (підпис) Стоянов Ю. М. (прізвище та ініціали)
Нормоконтроль (підпис) Стоянов Ю. М. (прізвище та ініціали)
Завідувач кафедри (підпис) Петрик М. Р. (прізвище та ініціали)
Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2024

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет Комп'ютерно інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра Програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р
(підпис) (прізвище та ініціали)
«__» _____ 2024 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Шиденку Ярославу Івановичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка емулятору ігрової платформи NES засобами мови програмування C#

Керівник роботи к.т.н., ст. викладач каф. ПІ Стоянов Ю. М.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 2024 року № _____

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи Предметна область, технічне завдання, вимоги та специфікація, програмне рішення

4. Зміст роботи (перелік питань, які потрібно розробити): Вступ. Розділ 1. Дослідження предметної області та завдання. Розділ 2. Опис та реалізація програмного продукту. Розділ 3. Тестування та можливі покращення емулятору. Розділ 4. Безпека життєдіяльності та основи Безпека життєдіяльності та основи охорони праці. Висновок.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів): Ілюстративні зображення, знімки екрану з проробленою роботою, схеми, діаграми.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	к. т. н. доцент каф. МТ Лазарюк В. В.		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи		
2.	Пошук інформаційних джерел по темі кваліфікаційної роботи		
3.	Знайомство з платформою NES та її компонентами		
4.	Розробка шару доступу до фізичної та оперативної пам’яті		
5.	Реалізація процесорної логіки платформи		
6.	Реалізація логіки PPU (Picture Processing Unit)		
7.	Розробка ігрового контролеру		
8.	Тестування емулятору на ігрових додатках		
9.	Виконання завдання до підрозділу «Безпека життєдіяльності»		
10.	Виконання завдання до підрозділу «Основи охорони праці»		
11.	Підготовка презентації та оформлення кваліфікаційної роботи		
12.	Нормоконтроль		
13.	Перевірка на плагіат		
14.	Попередній захист кваліфікаційної роботи		
15.	Захист кваліфікаційної роботи		

Студент _____ Шиденко Я. І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Стоянов Ю. М.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Розробка емулятора ігрової приставки NES з використанням мови програмування C# // Кваліфікаційна робота освітнього рівня «Бакалавр» // Шиденко Ярослав Іванович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-42 // Тернопіль, 2024 // С. – 80, рис. – 36, табл. – 0, кресл. – 0, додататки. – 2, бібліогр. – 16.

Ключові слова: емуляція, ігровий додадок, NES, C#, JetBrains Rider, OpenTK, VCS, Rendering, Game

Дана кваліфікаційна робота у деталях розглядає процес створення програмного додатку, який емулює роботу ігрової платформи NES. Основними цілями роботи були успішний запуск обраних ігрових додатків на створеній системі та отримання безцінного досвіду безпосередньо під час розробки.

У першому розділі кваліфікаційної роботи розглянуто предметну область в цілому, проведено пошук доступної інформації за даною тематикою, розглянуто ігрову платформу NES та те, з яких компонентів вона складається. Також був створений загальний план роботи над проєктом та усвідомлено роботу системи в цілому.

У другому розділі кваліфікаційної роботи описано та реалізовано сам програмний продукт. Даний розділ дає зрозуміти, як спроектований емулятор, з яких компонентів він складається, як виглядає його структура і чим вона відрізняється від оригінальної ігрової платформи.

У третьому розділі кваліфікаційної роботи розглянуто готовий програмний продукт, проведено тестування ігрових додатків безпосередньо на створенному емуляторі та складено план його подальшого розвитку.

ANNOTATION

Development of a NES game console emulator using the C# programming language // Bachelor's qualification work // Shydenko Yaroslav Ivanovych // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, group SP-42 // Ternopil, 2024 // P. – 80, fig. – 36, tabl. – 0, drawings – 0, addition – 2, literature – 16.

Keywords: emulation, gaming application, NES, C#, JetBrains Rider, OpenTK, VCS, Rendering, Game

The qualification work in detail examines the process of creating a software application that emulates the operation of the NES gaming platform. The main objectives of the work were the successful launch of selected gaming applications on the created system and the acquisition of invaluable experience during the development process.

The first section of the qualification work examines the subject area as a whole, conducts a search for available information on this topic, and reviews the NES gaming platform and its components. Additionally, a general project plan was created, and an understanding of the overall system operation was achieved.

The second section of the qualification work describes and implements the software product itself. This section provides an understanding of how the emulator is designed, its components, its structure, and how it differs from the original gaming platform.

The third section of the qualification work examines the completed software product, tests gaming applications directly on the created emulator, and outlines a plan for its further development.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

OpenTK (Open Toolkit) – фреймворк для роботи з OpenGL, OpenAL і OpenCL, який використовується для створення графічних і мультимедійних додатків.

C# – об'єктно-орієнтована мова програмування, розроблена компанією Microsoft для платформи .Net.

.Net – програмна платформа, яка забезпечує підтримку багатьох мов програмування і є основою для створення різноманітних додатків.

PPU (Picture Processing Unit) – процесор графічних даних, який відповідає за обробку і виведення зображень на екран.

CPU (Central Processing Unit) – центральний процесор, головний елемент комп'ютера, що виконує обчислення і керує роботою інших компонентів.

Rendering - процес створення зображень з моделі або сцени, який використовується в графічних додатках і іграх.

API (Application Programming Interface) – інтерфейс програмування додатків, набір інструментів і протоколів,

який дозволяє різним програмам взаємодіяти між собою.

OpenGL (Open Graphics Library) – відкрита бібліотека для роботи з графікою, яка забезпечує доступ до функцій апаратного прискорення для рендерингу 2D та 3D графіки.

Git – розподілена система керування версіями, яка дозволяє розробникам відстежувати зміни в коді і спільно працювати над проектами.

Rom (Read Only Memory) постійна пам'ять, що використовується для зберігання мікропрограмного забезпечення і даних, які не змінюються під час роботи пристрою.

Mapper – мікросхема, що керує доступом до пам'яті і розширює можливості системи, дозволяючи використовувати більшу кількість адресованої пам'яті та додаткові функції.

ЗМІСТ

ВСТУП

Розваги та індустрія комп’ютених ігор відіграють величезну роль на даному етапі розвитку суспільства. Вони дозволяють відволіктися від повсякденних турбот і стресів, надаючи можливість розслабитися та насолодитися часом для себе. Проте, така можливість не завжди присутня, оскільки на сучасних комп’ютерах не завжди можна пограти у щось таке, що ніколи не мало на меті працювати на новітніх платформах. Однією із таких систем є Nintendo Entertainment System, домашня ігрова консоль, яка була актуальною понад 30 років тому. За весь період її існування було створено понад 1000 ігор, які дозволяли зрозуміти, наскільки різноманітними та унікальними вони можуть бути. Робота над таким великим проєктом дає можливість ентузіастам та новому поколінню спробувати себе у ролі гравця 20 століття та побачити, наскільки сильно ігрова індрустрія змінилася з того часу.

Ще однією важливою складовою проєкту можна виділити практичність знань, отриманих у ході розробки програмного продукту. Розробка платформи для емуляції ігрової приставки NES є корисною для мене, як для розробника, оскільки вона включає в себе різноманітні етапи, що стосуються різних компонентів системи. Цей процес дозволяє отримати розуміння того, як ці компоненти взаємодіють між собою та як система функціонує в цілому. Створення такого програмного продукту сприятиме розвитку навичок програмування не лише на поверхневому, а й на більш глибокому рівні. Планується, що реалізований проєкт у майбутньому буде використовуватися гравцями та ентузіастами.

Програмне забезпечення буде створене за допомогою мови програмування C#. Для відтворення графіки та зчитування вхідних даних від користувача буде використана бібліотека OpenTK. У якості системи контролю версій буде використано систему Git, оскільки це сучасний та доволі простий інструмент, що дозволяє безпечно та комфортно працювати над проєктом, зберігаючи всю його історію. У якості клієнту буде використано застосунок SourceTree.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАВДАННЯ

1.1 Огляд предметної області

Історія відеоігор є доволі обширною. Протягом кількох десятиліть технології розвивалися, і це дозволяло створювати цікаві та різноманітні продукти, проте не варто забувати, з чого все починалося та як зароджувалася індустрія відеоігор. На мою думку, люди швидко зрозуміли, що обчислювальні машини можуть бути використані не лише у якості робочого інструменту, а й як те, що може використовуватися у розважальних цілях. Оскільки дана кваліфікаційна робота фокусується на ігровій платформі NES – доцільно буде розглядати такі ігрові системи, які передували її створенню.

Одними з перших ігрових систем були так звані аркадні автомати. Вони представляли собою стійку, у якій знаходилася вся апаратура, включаючи екран та систему керування. Деякі автомати могли пропонувати лише одну гру, яка була вбудована безпосередньо у пристрій. Однак були також автомати з можливістю зміни ігор, де ви могли вибирати з декількох запропонованих варіантів, що були завантажені у пристрій. Такі автомати надавали більшу різноманітність для гравців і зазвичай були популярнішими.

У 1972 році світ побачила перша домашня ігрова приставка під назвою Magnavox Odyssey. Вона могла відобразити на екрані лише 3 квадратних точки, і в залежності від того, яка ігрова картка використовувався у момент гри, вони могли міняти свою поведінку. Всього таких ігор для приставки було створено 28. Саме ця ігрова платформа надихнула компанію Atari на розробку відомої на весь світ гри Pong.

Рисунок 1.1 – Ігровий автомат, створений для гри Pong (1972)

Прогрес не стояв на місці, та вже через кілька років, у 1977 вийшла популярна ігрова консоль Atari 2600. Це була система другого покоління. Вона протягом доволі тривалого терміну залишалася однією з найпопулярніших платформ для гри, у неї було набагато більше можливостей, ніж у попередників і починаючи з цієї ігрової платформи з’явилася велика кількість класичних ігор, таких як Pac-Man та Breakout. Це була перша ігрова система, яка використовувала картриджі у якості носіїв даних.

Рисунок 1.2 – Гра Pac-Man на платформі Atari 2600

У 1983 році з’явилася нова ігрова приставка третього покоління Famicom, яку згодом перемейнували на Nintendo Entertainment System. Найбільшими її перевагами над системами минулого покоління були кращі технічні характеристики та дуже великий перелік ігор, які створювалися для даної консолі. Талісманом корпорації Nintendo та приставки NES став персонаж Mario з гри Super Mario Bros, яка була а той час бестселлером та могла продемострувати апаратні можливості ігрової системи. Метою даної кваліфікаційної роботи є дослідження та написання емулятору саме цієї ігрової системи, оскільки вона, на мою думку, зробила вагомий внесок у розвиток ігрової індустрії та ігрові додатки, які виходили на дану платформу, стали у своїй більшості класичними.

1.2 Огляд компонентів ігрової платформи

Перш за все потрібно зрозуміти, що являє собою емуляція ігрової платформи. Це доволі складний процес, який розбитий на велику кількість етапів, які у кінцевому результаті є тісно пов’язаними та працюють, як одне ціле. У контексті даної кваліфікаційної роботи, емулятором ігрової платформи NES можна назвати таке програмне забезпечення, яке відтворює апаратну логіку оригінальної консолі. З основних компонентів можна виділити центральний процесор, графічний процесор, систему картриджів, ігровий контролер та аудіо процесор.

- Центральний процесор (CPU) відповідає за виконання інструкцій програмного коду, контролюючи роботу всіх інших компонентів.
- Графічний процесор (PPU) займається обробкою відеосигналів, відтворюючи зображення на екрані, яке відповідає оригінальному вигляду ігор.
- Система картриджів забезпечує зчитування ігрових даних, які зберігаються на фізичних носіях, і передає їх на обробку іншим компонентам.

- Ігровий контролер емулює введення від гравця, дозволяючи використовувати сучасні пристрої вводу для взаємодії з емулятором так, як це було на оригінальній консолі.
- Аудіо процесор відтворює звукові ефекти та музику, що додає автентичності геймплею.

Варто зауважити, що усі ці компоненти повинні працювати разом синхронно, щоб створити точну копію роботи оригінальної системи NES. Це вимагає ретельного тестування та глибокого розуміння архітектури, аби забезпечити високу точність емуляції і сумісність з різними іграми. Крім того, емулятор повинен бути ефективним з точки зору використання ресурсів, щоб забезпечити плавний ігровий процес на сучасних пристроях. Це завдання потребує глибокого розуміння апаратної архітектури NES та значного технічного досвіду, що робить його цікавою і водночас складною темою для кваліфікаційної роботи.

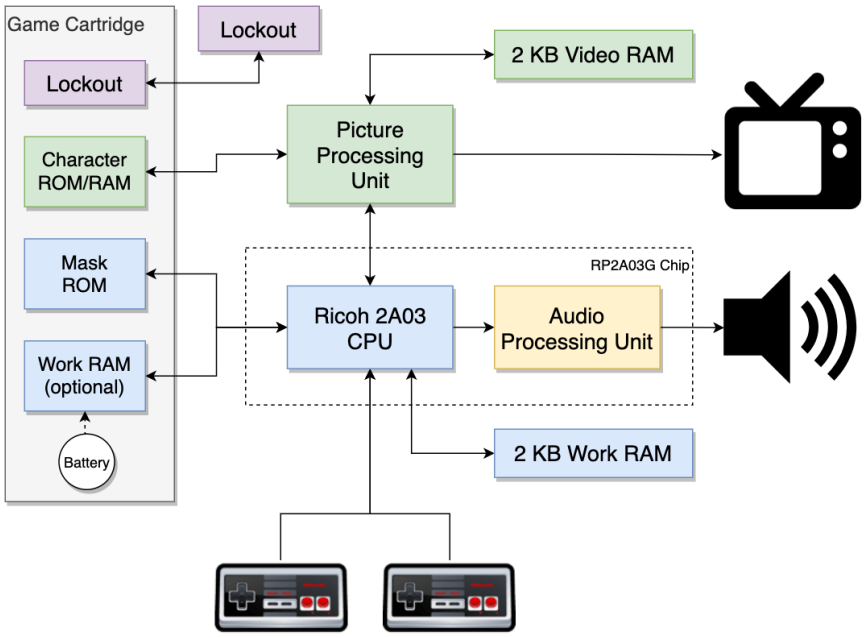

Рисунок 1.3 – Схема архітектури системи NES

На рисунку 1.3 зображено детальну схему взаємодії компонентів системи між собою, що дозволяє зрозуміти зв’язок між ними та те, яким чином потрібно підходити до проєктування системи [7].

1.3 Огляд цілі проєкту

Як було згадано вище – кілька десятків років тому NES вважалася однією з найбільш популярних та просунутих ігрових платформ. Сотні тисяч гравців по всьому світу мали можливість спробувати та відчути щось нове у сфері ігор, проте станом на сьогодні це не завжди так. Оскільки класичних ігор для неї було створено багато – у ентузіастів є потреба у тому, аби була можливість відтворити ці ігри на новітніх комп’ютерах. Цей проєкт має на меті зробити свій внесок у збереження цифрової спадщини, забезпечуючи доступ до ігор, які могли б бути втрачені для майбутніх поколінь. Оскільки для написання повноцінного емулятору з врахуванням усіх доступних ігор потрібно дуже багато часу – продукт можна буде вважати реалізованим після того, як на емуляторі буде можливість пограти у наступні ігрові додатки: Super Mario (1983), Battle City (1985), Donkey Kong (1983).

Отже, основною ціллю даної кваліфікаційної роботи є створення такого програмного рішеня, яке засобами емуляції зможе відтворювати значну кількість випущених для платформи NES ігор.

1.4 Підбір джерел інформації та інструментів розробки

Для пошуку джерел інформації, що будуть використовуватися у рамках даної кваліфікаційної роботи, були встановлені певні критерії, яким вони повинні відповідати, а саме достовірність інформації та якість доступного матеріалу. Після перегляду доступної інформації, було знайдено та обрано кілька ресурсів, на основі яких буде реалізовано емулятор системи NES.

Оскільки у мене, як у розробника, є значний досвід написання програм за допомогою технології .NET [1] та мови програмування C# [2] – було вирішено, що вони будуть вибрані у якості основних ісntрументів розробки. Варто зазначити, що C# – це сучасна об'єктно-орієнтована мова програмування, розроблена компанією

Microsoft як частина платформи .NET. Вона поєднує в собі простоту та продуктивність, підтримуючи безпеку типів і автоматичне управління пам'яттю. C# [2] використовується для розробки різноманітного програмного забезпечення, включаючи десктопні та веб-додатки, ігри, а також мобільні додатки. Мова має велику стандартну бібліотеку та інтеграцію з іншими технологіями Microsoft, що робить її потужним інструментом для розробників [1].

Для реалізації відображення графічного контенту та зчитування користувацького вводу було обрано фреймворк OpenTK [3], який являє собою набір бібліотек, що надають доступ до графічних і звукових API, таких як OpenGL [12], OpenCL і OpenAL на платформі .NET. Він був створений для розробки кросплатформенних мультимедійних додатків, включаючи ігри та наукові візуалізації. OpenTK [3] спрощує взаємодію з низькорівневими графічними та аудіо функціями завдяки простим у використанні обгорткам. Враховуючи сказане вище – можна дійти до висновку, що даний фреймоврок буде доволі хорошим вибором у якості інструменту для написання емулятору [2].

Рисунок 1.4 – Логотип фреймворку OpenTK

У якості інтегрованого середовища розробки обрано JetBrains Rider [4], оскільки це сучасне та доволі просунуте рішення, яке дозволяє розробляти складні програмні додатки, використовуючи мову програмування C# [2]. Для управління версіями проєкту було обрано систему контролю версій Git [5], яка відома своєю надійністю та можливостями. Для зручності роботи з репозиторіями Git [5]

вирішили використовувати клієнт SourceTree [6], який забезпечує інтуїтивно зрозумілий графічний інтерфейс і потужний функціонал для розробників.

1.5 Складання плану розробки інформаційної системи

Отже, для того, аби розпочати розробку програмного додатку, який буде емулювати роботу ігрової приставки NES – потрібно скласти план, який дозволить чіткіше розуміти те, на які етапи можливо розділити цей процес. На мою думку, варто виділити наступні пункти:

1. Знайомство з ігровою системою, її специфікацією та технічними аспектами емуляції.
2. Аналіз джерел інформації, пошук технічної документації та статей, що описують апаратну архітектуру системи, центрального процесору, графічного процесору тощо. Створення головного класу, який буде початковою точкою для написання емулятору.
3. Реалізація каркасу програмного продукту, детальне знайомство з його ключовими компонентами, реалізація контролеру картриджів та знайство з мапперами, роботою адресної шини, оперативної пам’яті та стеку. Знайомство з типами адресації. Написання класу, що відповідає за центральний процесор та створення логіки його ініціалізації.
4. Знайомство з шаблоном проєктування “Update Loop”, його реалізація у рамках проєкту. Синхронізація центрального та графічного процесорів. Підготовка до написання процесорної логіки.
5. Реалізація логіки центрального процесору.
6. Реалізація логіки генератору зображень (PPU).
7. Тестування емулятору.

РОЗДІЛ 2. ОПИС ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Знайомство із апаратними компонентами ігрової системи NES

У час, коли створювалася ігрова приставка NES – було два головних аналогових телевізійних формати, а саме NTSC та PAL. В основному вони відрізнялися частотою струму у розетці. У першому варіанті вона становила 60 Гц, коли у другому – 50. Це безпосередньо впливало на частоту оновлення телевізійних екранів та на деякі апаратні обмеження. Наприклад, ігри на приставці, що розроблена для формату NTSC, працювали на частоті 60 кадрів за секунду, підтримували роздільну здатність 256 на 240 пікселів та їх процесор працював на частоті 1.79 MHz (у той час, коли на PAL вона була 1.66 MHz). Також варто зазначити те, що ігри, випущені на різні версії консолі, могли працювати на них по-різному та з різною швидкістю. Враховуючи сказане – було вирішено взяти формат NTSC за основу на будувати емулятор використовуючи притаманні для властивості приставки [7].

Для того, аби розпочати роботу над емулятором – потрібно зрозуміти, з якими компонентами доведеться працювати. Основою системи був процесор Ricoh 2A03, який вважався майже точною копією MOS Technology 6502, за вийнятком деяких модифікацій та нюансів. Це 8-розрядний процесор із 16-розрядною адресною шиною, завдяки чому може адресувати 64 КБ пам’яті. У нього фіксована частота (1.79 MHz) та час виконання інструкцій у тактах. У рамках оптимізації передбачена велика кількість режимів адресації, які програмісти можуть використовувати під час написання власних програм [7].

Рисунок 2.1 – Мікропроцесор Ricoh 2A03

За генерацію зображення відповідає Ricoh 2C02, також відомий, як PPU (Picture Processing Unit). Це окремий мікропроцесор, який працює на частоті, що рівно у 3 рази перевищує частоту процесора (5.37 MHz). Також він має власну відеопам’ять (2 КБ) та 256 байт пам’яті, у якій зберігається інформація про спрайти, які повинні бути відмальовані на екрані (до 64 спрайтів). У графічного процесора є доступ до палітри кольорів, що складається із 64 елементів. Як вже було згадано раніше – враховуючи формат NTSC, стандартним розширенням консолі, на яке орієнтується графічний процесор, вважається 256 x 240 пікселів. За сихнронізацію компонентів платформи відповідає системний годинник, відносно якого обраховується частота центрального та графічного процесорів. У даному випадку його частота становить ~21.47 MHz. У рамках даної роботи компоненти будуть синхронізуватися згідно частоти центрального процесору, тому імплементацію системного годиннику можна вважати не обов’язковою, знаючи про цільову частону центрального процесору [7].

Також важливим компонентом NES є система контролерів, що дозволяє взаємодіяти з консоллю та іграми. Стандартні контролери NES мають чотири кнопки дій (A, B, Select, Start) і хрестовину (D-pad) для навігації. До консолі їх може бути підключено декілька, проте у даному проєкті буде використаний лише один. Варто виділити систему картриджів та мапперів, що взаємодіють між собою для того, аби підтримувати ігри різного розміру та розширювати можливості адресації ігрової приставки [7].

2.2 Робота над архітектурою та базовими компонентами проєкту.

Робота над емулятором платформи NES розпочинається зі знайомства з форматом бінарних файлів типу iNes (.nes). Цей формат був розроблений ще у далеких 1990-х роках і за весь час свого існування де-факто став стандартом дистрибуції ігрових додатків. Як і будь-який з існуючих бінарних форматів, він

складається з кількох послідовних частин з фіксованим розміром, у яких зберігаються дані, що стосуються гри, яку репрезентує обраний файл. Розгланемо основні частини файлу:

- Заголовок (16 байт) містить важливу інформацію про те, яким чином потрібно зчитувати інформацію з файлу. Байти 0-3 містять у собі константу “NES”; байт 4 відповідає за розмір PRG ROM (у одиницях розміром 16 КБ); байт 5 відповідає за розмір CHR ROM (у одиницях розміром 8 КБ); байти 6-10 містять у собі додаткову інформацію про те, яка версія формату використовується тощо;
- Дані PRG ROM (16384 байт, помножені на число, отримане у 4 байті заголовку), містять безпосередньо команди, які виконуються на процесорі;
- Дані CHR ROM (8192 байти, помножені на число, отримане у 5 байті заголовку), містять ігрову графіку;
- Не використаний зсув (байти 11-15) [7].

Оскільки у оригінальної ігрової приставки були певні обмеження, пов’язані з можливістю процесора адресувати лише 64 КБ пам’яті – розробникам доводилося обходити це обмеження, використовуючи такий принцип, як перемикання банків. Банками називаються блоки пам’яті, на які поділена інформація у картриджі (зазвичай по 8 або 16 КБ), і мікроконтроллер всередині картриджу має можливість керувати тим, який банк пам’яті буде відображений процесору через адресну шину. Це перемикання здійснюється шляхом запису певних значень у спеціальні регістри мікроконтроллера. Такі конфігурації картриджів називають “мапперами”, оскільки вони можуть описати, яким чином адреси вказують на фізичну пам’ять. У рамках даної кваліфікаційної роботі, ми зосередимося на кількох основних маперах, а саме: маппері під індексом 0, який також називають “NROM”, та маппером з індексом 2, який має назву “UxROM”. Перший не потребує додаткових маніпуляцій для роботи, оскільки являє собою звичайний картридж без будь-яких мікросхем всередині. Другий, у свою чергу, має можливість перемикання банків та дає можливість системі NES працювати з файлами різного розміру [8]. Також буде спроєктовано клас, який є сутністю, що містить інформацію про тип мапперу.

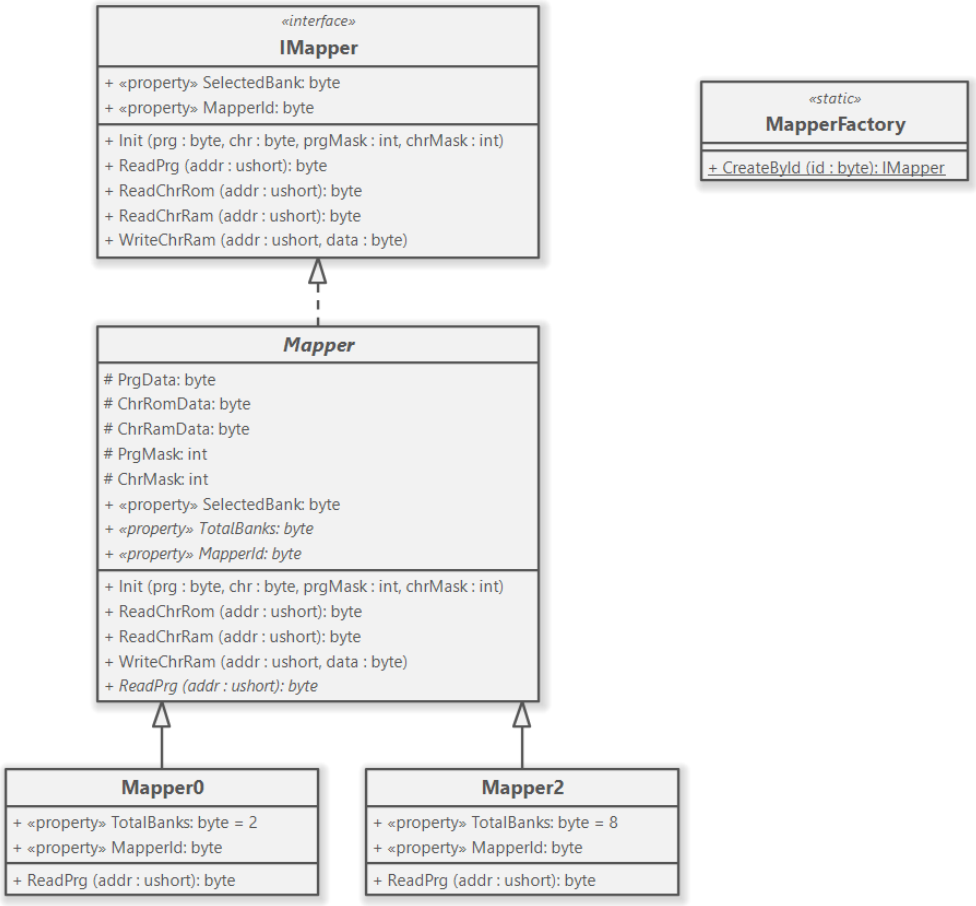

Рисунок 2.2 – Загальна діаграма класів сутностей мапперів

Для опису згаданої вище логіки був створений клас, який у якості вхідного аргументу приймає шлях до бінарного файлу формату “.nes”, зчитує потрібну інформацію та створює новий об’єкт класу Rom.cs, який містить в собі інформацію у вигляді прапорців та двох масивів даних з функціями, які дозволяють їх прочитати. Також було створено інтерфейс типу IMapper (рисунок 2.1), що містить у собі опис мапперів та компонентів, з яких вони складаються. Він імплементовується у абстрактному класі типу Mapper, який містить спільку для всіх мапперів конфігурацію. Наслідують даний клас вже безпосередньо реальні реалізації мапперів, у нашому випадку це маппери 0 та 2. Кожен з них перевизначає певні параметри, які в майбутньому будуть використовуватися для написання відповідної логіки. Варто сказати, що у них зберігається обраний банк пам’яті та загальна кількість банків, оскільки це є необхідною складовою будь-якого мапперу.

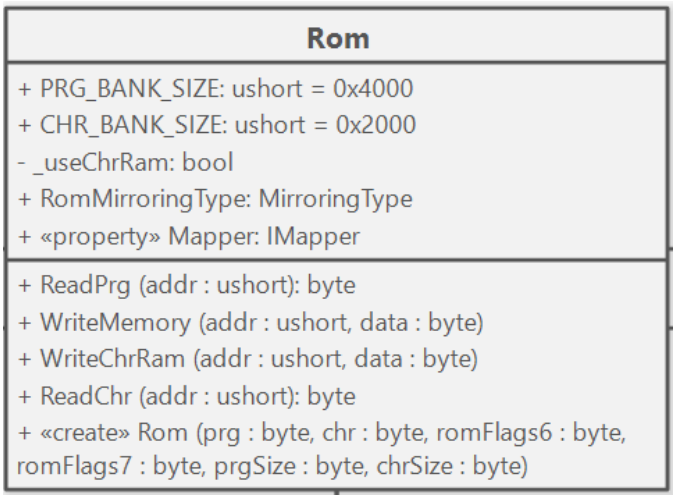

Рисунок 2.3 – Діаграма класу Rom.cs

Процесорні інструкції розташовані безпосередньо у PRG ROM, проте виконувати їх потрібно не з самого початку файлу. Для того, аби розпочати написання процесорної логіки – при завантажені ігрового образу потрібно ініціалізувати процесор. Це робиться за допомогою векторів переривань, які являють собою спеціальні адреси у пам’яті, що використовуються процесором для визначення, куди передати керування при виникненні переривання. Потрібний нам вектор знаходиться за адресами $FFFC-$FFFD та відповідає за скидання процесора, яке відбувається під час запуску системи або її перезапуску. Згідно специфікації, код ініціалізації повинен міститися у всіх ігрових додатках і від розробників очікується, що буде виконана певна логіка, оскільки це важлива частина програмного коду, основною метою якої є налаштування початкових умов роботи апаратних компонентів системи. Здебільшого у коді ініціалізації реалізоване очищення оперативної пам’яті, налаштування режимів відображення тощо [7].

Для того, аби у процесора була можливість комунікувати з іншими компонентами системи та зрозуміти, куди саме вказує вектор переривання – потрібно створити клас, який буде описувати логіку цієї комунікації. Тобто, виникає необхідність у можливості читання та запису даних для обміну інформацією. Так як зв’язок між діапазонами пам’яті та системними компонентами відомий, оскільки він реалізований на апаратному рівні, проблем під час реалізації такого фукнціоналу виникнути не повинно. Враховуючи ті компоненти, які

використовуються у рамках даної кваліфікаційної роботи, це відношення виглядає наступним чином:

- $0000–$07FF вказують на 2КБ оперативної пам’яті.
- 3 блоки по 2КБ у діапазоні $0800–$1FFF є дзеркалами, які також вказують на оперативну пам’ять.
- $2000–$2007 вказують на регістри PPU, завдяки чому процесор має можливість комунікувати з графічним процесором.
- $4000–$4017 вказують на регістри аудіопроцесору та контролеру вводу-виводу.
- $8000–$FFFF вказують на пам’ять картриджу [7].

Отже, враховуючи сказане вище, був створений клас MainBus.cs, який містить у собі методи для читання та запису по вказаній адресі. У архітектурному плані, цей клас повинен містити у собі посилання на усі інші компоненти, оскільки він безпосередньо приймає участь у обробці даних, які у них знаходяться. За домогою switch-кейсу можна визначити, до якого компоненту відноситься адреса, вказана у вхідному параметрі реалізованих функцій. На рисунку 2.4 наведено приклад читання одного байту даних за 16-бітною адресою. Функція для запису даних має схожу логіку, проте у якості аргумента додатково приймає байт, який потрібно записати у пам’ять.

```
//16-bit addressing mode.
82 usages
public byte ReadMemoryByAddr(ushort address)
{
    switch (address)
    {
        case <= 0x1FFF:
            return _ram.Read(address);
        case <= 0x2007:
        {
            _ppu.CatchUpToCpuCycle(_cpu.CurrentCycle);
            return _ppu.ReadRegister( index: (byte)(address - 0x2000));
        }
        case 0x4016:
            return _controller.ReadNextButton();
        case >= 0x8000:
            return _rom.ReadPrg(address);
    }
    return 0;
}
```

Рисунок 2.4 – Реалізація логіки читання даних за адресою у класі MainBus.cs

Наступним важливим завданням є робота над оперативною пам’яттю. Потрібно зрозуміти, що вона являє собою та яким чином процесор комунікує з нею. У першу чергу варто зазначити, що вона поділена на рівні частини розміром 256 байт, які також називають сторінками пам’яті. Перша сторінка оперативної пам’яті є особливо, називається вона “Zero Page”. До неї відносяться адреси $0000-$00FF і пізніше на ній ще буде акцентовано увагу через однойменний тип адресації пам’яті. Також варто пригадати про такий важливий елемент, як стек. Це структура даних, яка працює за принципом LIFO (Last-In, First-Out) та над нею можна виконувати декілька операцій: push для того, аби додати елемент; pull для того, аби отримати елемент і видалити його зі стеку; peek для того, аби отримати елемент, проте не видаляти його зі стеку. Дані операції підтримуються безпосередньо самим центральний процесором і у кожної з них є своя відпоідна інструкція. У процесорі Ricoh 2A03, який використовує архітектуру MOS 6502, стек імплементований за допомогою буферу розміром 256 байт, що займає весь адресний простір другої сторінки пам’яті. Вказівник стеку дає зрозуміти, де знаходиться останній елемент, який був доданий у стек і являє собою вже згаданий вище процесорний регістр під назвою SP (Stack Pointer). Варто сказати, що стек росте зверху вниз і вважається спадаючим. Тобто перший його елемент буде знаходитися на верхівці другої сторінки оперативної пам’яті, і з кожним наступним доданим елементом вказівник стеку буде зменшуватися і, відповідно, збільшуватися, якщо елемент буде видалено.

Існує 2 типи інтерпретації вказівнику стеку, а саме “Empty Stack” та “Full Stack”. Перший вказує на частину пам’яті, куди буде записаний наступний елемент, у той час коли другий вказує на самий верхній елемент стеку. У MOS 6502 використовується перша інтерпретація. Тобто, інформація спочатку записується, після чого значення у вказівнику стеку зменшується на 1 (росте вгору). З отримуванням даних ситуація протилежна: спочатку потрібно інкрементувати вказівник стеку, після чого прочитати значення через раніше створений клас MainBus.cs з оперативної пам’яті. За ініціалізацію вказівнику стеку відповідає код

ігрової програми, який виконується після встановлення початкової адреси у регістр PC [7].

Рисунок 2.5 – Функції для роботи зі стеком

Одним з наступних кроків у рамках даної кваліфакаційної роботи є знайомство з типами адресації, які підтримує процесор ігрової платформи. Всього їх існує 12, проте тих, які використовують оперативну пам’ять, є лише дев’ять [9]. Серед них :

- Абсолютний (Absolute) режим адресації вказує на адресу пам’яті, з якою ми будемо працювати, у наступних двох байтах після коду інструкції.
- Відносний (Relative) режим адресації використовується лише у операціях переходу (branch), що дозволяють змінювати потік виконання. Одразу після коду інструкції потрібно прочитати 1 байт, що являє собою зсув, який буде застосовано за певних умов в залежності від інструкції. Цей байт представляє собою число зі знаком від -128 до 127 включно, що дозволяє пересуватися як вперед, так і назад.
- Zero Page має принцип, схожий до Absolute, проте з можливістю адресувати лише перших 256 байт оперативної пам’яті, тобто першу сторінку. Головними його перевагами є менший розмір та менша кількість процесорних інструкцій, необхідних для виконання.
- Indirect використовується лише однією інструкцією (JMP). Інструкція займає 3 байти, і 2 та 3 байти вказують адресу, за якою знаходиться інша адреса, з якою потрібно працювати.

- Absolute Indexed (X, Y) працює аналогічно до режиму Absolute, проте до адреси ще додається вміст процесорного регістру X або Y.
- Zero Page Indexed (X, Y) працює так само, як і Absolute Indexed, проте у рамках першої сторінки оперативної пам'яті. Варто зауважити, що результат операції додавання не може виходити за її межі. Наприклад, якщо адрес дорівнює $0255, а регістр X або Y = 1, то у результаті це вкаже нам на адресу $0000, що називається переповненням з обгортанням.
- Indexed Indirect використовується лише з регістром X. Другий байт інструкції вказує на комірку, що знаходиться на першій сторінці пам'яті. До неї необхідно додати вміст процесорного регістра X, що у результаті буде вказувати на молодший байт цільової адреси. Для того, аби отримати її повністю – потрібно прочитати також старший байт, який знаходиться у наступній комірці пам'яті та об'єднати їх у 16-бітне число.
- Indirect Indexed вважається одним із найпоширеніших типів адресації та використовується лише з регістром Y. Для того, аби отримати реальну адресу, з якою нам потрібно працювати – необхідно прочитати старший і молодший байти, що знахотяться на першій сторінці оперативної пам'яті, після чого з'єднати їх у 16-бітне число та додати до нього вміст регістру Y. Для отримання кінцевого результату потрібно до фінальної адреси додати вміст процесорного регістру Y. Реалізація даного типу адресації продемонстрована на рисунку 2.6.

```
3 usages
private ushort ReadAddrIndirectY()
{
    var zeroPageAddr :byte = _bus.ReadNextByte();
    var firstAddr :byte = _bus.ReadMemoryByAddr(zeroPageAddr);
    var secondAddr :byte = _bus.ReadMemoryByAddr( address: (byte) (zeroPageAddr + 1));
    var finalAddr = (ushort) (firstAddr | (secondAddr << 8));
    CheckIfPageCrossed( oldAddr: finalAddr, newAddr: (ushort) (finalAddr + Y));
    return (ushort) (finalAddr + Y);
}
```

Рисунок 2.6 – Принцип роботи Indirect Indexed адресації

Тепер потрібно розглянути ті типи адресації, які не використовують оперативну пам’ять під час роботи. Усього їх є 3:

- Accumulator працює безпосередньо з однойменним регістром, записуючи туди результат виконаної роботи. Загальний розмір – 1 байт.
- Immediate використовує 1 байт даних, що йде одразу після коду інструкції, як число і проводить певну операцію з цим числом над процесорним регістром A (Accumulator) в залежності від інструкції. У мові асемблера для вказання того, що у якості операнда використовується число, а не адреса – перед ним потрібно поставити знак “#”.
- Implied інструкції не використовують додаткову пам’ять взагалі, оскільки вони не пишуть та не читають дані і здебільшого використовуються коли потрібно оновити прапорці процесора або виконати операції, які не потребують роботи з даними. Прикладом такої інструкції є “Clear Carry Flag” (CLC).

Враховучи сказане вище, стало відомо, яким чином можливо оперувати адресами та даними, використовуючи створений клас з відповідною логікою. Проте для того, аби розпочати написання логіки центрального процесору – потрібно познайомитися із його внутрішніми регістрами. Вони використовуються для арифметичних, бінарних та багатьох інших операцій і керуються безпосередьо процесорними інструкціями. Ricoh 2A03 має декілька внутрішніх регістрів, а саме:

- Accumulator – використовується для арифметичних та логічних операцій.
- X (Index Register X) – використовується для деяких типів адресації пам’яті та як лічильник у циклах.
- Y (Index Register Y) – використовується для тих самих цілей, що й регістр X.
- PC (Program Counter) – вказує на адресу наступної інструкції, яку має виконати процесор.
- SP (Stack Pointer) – регістр, який вказує на верхній елемент стеку.
- P (Processor Status) – описує поточний стан процесора за допомогою різноманітних прапорців [9].

Кожен з вказаних регістрів є 8-бітним, окрім PC, який являється 16-бітним регістром. Для їх опису будуть використовуватися типи byte та ushort відповідно. Після опису даних регістрів потрібно додати константу, яка репрезентує частоту процесора, а саме 1.79MHz. Варто пригадати, що цю частоту задає “Master Clock”, проте немає необхідності у настільки точній емуляції, оскільки частосу процесору можна оприділити самостійно, спираючись на відомі дані. Пізніше вона буде використана емулятором для того, аби він працював з правильною швидкістю.

```
5 usages
public class Cpu
{
    public const long Frequency = 1790000;
    private const ushort StackAddr = 0x100;

    public long CurrentCycle;

    public byte Accumulator;
    public byte X;
    public byte Y;
    public byte StackPointer;
    public byte StatusFlags;
}
```

Рисунок 2.7 – Опис компонентів процесору у файлі Cpu.cs

Отже, повернімося до ініціалізації центрального процесору. Як було сказано вище, нам потрібно зрозуміти, куди посилається вектор переривання, який стосується адресів $FFFC-$FFFD. Для цього потрібно по-порядку прочитати молодший та старший байти, і з’єднати їх у 16-бітне число, яке буде представляти собою адрес, який повинен бути встановлений у регістр PC, що знаходиться у класі, який описує центральний процесор. Також за замовчуванням потрібно встановити певні процесорні прапорці та врахувати кількість процесорних циклів, витрачену на таку операцію [9].

```
private void ResetCpu()
{
    var firstByte = _bus.ReadMemoryByAddr(address: 0xFFFC);
    var secondByte = _bus.ReadMemoryByAddr(address: 0xFFFD);
    _cpu.Pc = (ushort)(firstByte | secondByte << 8);
    _cpu.CurrentCycle += 7;
    _cpu.StatusFlags = 0x24;
}
```

Рисунок 2.8 – Встановлення PC на початкову адресу

Тепер, коли у нас створений клас, який являє собою об’єкт, що відповідає за головний компонент системи та ініціалізовний програмний лічильник – потрібно реалізувати програмний цикл, який буде емулювати його роботу та з певною частотою виконувати процесорні інструкції. Для цього у проєкті використовується фреймворк під назвою OpenTK [3], який реалізовує шаблон проєктування типу “Update Loop” [10], який також називається циклом подій. Його основною властивістю у рамках даного фреймворку є те, що у нас є головний, нескінченний цикл, який ітерується з певною заданою частотою, зчитує користувацькі дані та оновлює зображення на екрані. Оскільки емулятор, як і будь який графічний додаток, дотримується цих принципів – у рамках оптимізації з’являється можливість виконувати процесорні інструкції великими групами під час оновлення кадрів. Для цього потрібно створити клас EmulatorWindow.cs, який успадковується від класу фреймворку GameWindow.cs, після чого йому потрібно задати параметри ширини та висоти вікна, а також частоту оновлення у герцах. Оскільки базовий клас містить віртуальну функцію “OnUpdateFrame” – ми можемо перевизначити його у створеному похідному класі. Дана функція викликається фреймворком автоматично у рамках циклу подій в залежності від заданої частоти оновлення. Тобто, у нашому випадку, згадана функція буде викликатися 60 разів на секунду. Варто сказати, що також у функцію передається аргумент типу float, що являє собою час у мілісекундах, що пройшов з моменту попереднього оновлення і даному випадку це число завжди буде близьким до 16.6 мілісекунд. Під час кожного такого виклику нам потрібно оновлювати стан центрального процесору, тому для цього у головному класі Emulator.cs було створено функцію “Update”, у яку ми передаємо

різницю у часі між теперішнім та попереднім оновленнями, аби порахувати, скільки інструкцій за цей час нам потрібно виконати.

Варто пам’ятати, що кожна процесорна інструкція займає якийсь певний час для того, аби виконатися, і це залежить напряму від того, наскільки інструкція є складною та з чим вона взаємодіє. У центральному процесорі цей час вимірюється у процесорних циклах і те, скільки таких операцій процесор зможе виконати за певний проміжок часу залежить від того, яка у нього тактова частота. Чим більша частота – тим більше інструкцій може виконати процесор за відведений час. Рахувати процесорні цикли необідно для того, аби ввести поняття витраченого на певні операції часу та для того, аби мати можливість сихнронізувати роботу центрального та графічного процесорів.

```
1 usage
public void Update(double dt)
{
    var targetCycle :long = _cpu.CurrentCycle + (int)(Cpu.Frequency * dt);
    while (_cpu.CurrentCycle < targetCycle)
    {
        _cpu.ExecuteInstruction();
    }
}
```

Рисунок 2.9 – Логіка функції оновлення процесорного стану

На рисунку 2.9 зображено те, як повинно виглядати оновлення процесорного стану у рамках головного програмного циклу. Нам потрібно порахувати, скільки процесорних циклів пройде за період оновлення одного кадру. Для цього необхідно взяти цикл, на якому зараз знаходиться процесор, і додати до нього частоту процесора, помножену на часову різницю між оновленнями. Після даної операції можна виконувати наступну процесорну інструкцію, на яку вказує регістр PC до того моменту, доки локальний цикл процесора не наздожене необхідне значення, пораховане у змінній “targetCycle”.

Враховуючи сказане вище – можна розпочинати написання безпосередньо процесорної логіки. Для того, аби було видно прогрес під час розробки, інструкції було вирішено реалізовувати поступово у такому порядку, у якому вони записані у

бінарному файлі гри, з яким ведеться робота. На мою думку, це доволі зручний підхід, оскільки ми розробляємо логіку лише тих процесорних інструкцій, які нам потрібні. Якщо якась інструкція ще не реалізована – додаток повідомить про це через такий механізм мови C#, як виняткова ситуація. У якості ігрового додатку під час розробки було обрано проєкт Battle City (1985). Оскільки процесорні інструкції доволі часто можуть мати декілька типів адресації – у рамках даної кваліфікаційної роботи увага буде зосереджена безпосередньо на логіці самих інструкцій та на процесі їх взаємодії з оперативною пам’яттю та внутрішніми регістрами центрального процесору.

2.3 Реалізація логіки центрального процесору.

Написання та продумування логіки центрального процесору розпочинається зі створення функцій у класі Cpu.cs, який являє собою сутність, що описує логіку та функціонал головного компоненту системи. У попередньому підрозділі вже було описано декілька пунктів, що стосуються процесору та які є необхідними для того, аби мати можливість займатися безпосередньо розробкою, а саме:

- Описано процесорні регістри та ситуації, у яких вони використовуються.
- Реалізовано логіку скидання процесору та його встановлення на початкову адресу.
- Описано стек, його реалізацію та функції для роботи з ним.
- Описано механізм доступу до пам’яті, шини адресів та даних.
- Описано типи адресації.

Наступним етапом варто зрозуміти, що саме представляють собою процесорні інструкції, як вони повинні виконуватися у рамках емулятору і коли це взагалі повинно відбуватися. Процесорні інструкції - це базові команди, які виконуються центральним процесором. Вони складаються з операцій, таких як додавання, віднімання, переміщення даних та інші дії, які процесор використовує

для виконання програм. Кожна інструкція зазвичай має свій унікальний код операції (opcode) і, опціонально, операнди, які використовуються під час виконання інструкції [9]. У рамках емулятору вони повинні виконуватися так само, як це відбувалося б на реальному процесорі. Цей етап складається з декількох пунктів:

- Зчитування інструкції з пам’яті за допомогою програмного лічильника PC, що являє собою 16-бітний регістр процесора, який вже згадувався раніше. Він вказує на те, з якого місця потрібно читати наступну інструкцію.
- Розшифровка інструкції, яка складається зі зчитування коду операції та операндів, які йдуть одразу після нього, якщо такі є.
- Виконання процесорної логіки, визначеної кодом операції. В залежності від її типу, це може бути арифметична операція, скидання прапорців процесора та інші.
- Збереження результатів виконання інструкції у відповідному регістрі або у оперативній пам’яті.

Оскільки процесор працює безперервно та виконує ці кроки постійно – потрібно створити функцію, яка буде містити у собі описану логіку, і викликати її стільки разів, скільки це необхідно, враховуючи тактову частоту центрального процесору. Для цього у попередньому розділі був створений клас EmulatorWindow.cs, що містить у собі функцію “Update”, яка викликається потрібну кількість разів.

Отже, у класі Cpu.cs було створено функцію “ExecuteInstruction”, у якій реалізовано switch case, що дозволяє зчитувати код операції з пам’яті та одразу імплементувати логіку в залежності від нього. Всього таких кодів може бути 255, оскільки кожен код операції займає лише 1 байт пам’яті. Враховуючи цю інформацію – можна прочитати декілька перших інструкцій безпосередньо з PRG ROM та реалізувати їх логіку.

```
1 usage
public void ExecuteInstruction()
{
    var opcode :byte = _bus.ReadNextByte();
    switch (opcode)
    {
        //SEI (Set Interrupt Disable)
        case 0x78:
            AssignFlag( mask: StatusFlagsConstants.INTERRUPT_DISABLE,  condition: true);
            CurrentCycle += 2;
            break;
        //LDA (Load Accumulator)
        case 0xA9:
            Accumulator = _bus.ReadNextByte();
            UpdateStatusFlags(Accumulator);
            CurrentCycle += 2;
            break;
    }
}
```

Рисунок 2.8 – Логіка перших процесорних інструкцій

На рисунку 2.8 продемонстровано, яким чином виглядають інструкції та як виконується їх логіка. Наприклад, код операції $78 представляє собою інструкцію “SEI (Set Interrupt Disable)”, яка модифікує один біт у процесорному регістрі “Status Flags”. Для цього була також створена відповідна функція та список констант, які дозволяють визначити, що саме потрібно змінити та яким чином. У якості аргументу функція “AssignFlag” приймає маску байтового типу, яка вказує на те, який саме біт потрібно змінити. У даному випадку наша константа це “INTERRUPT_DISABLE”, яка у десятковому форматі представляє число 4, а у двійковому вона виглядає як 0b_100. Другим аргументом даної функції є параметр типу bool, який вказує, яким повинен бути біт, що вказаний за допомогою маски.

Процесорні прапорці оновлюються під час великої кількості операцій в халежності від результату їх виконання. Як можна бачити на рисунку 2.8, наступною інструкцією після “SEI” є “LDA (Load Accumulator)” з кодом операції $A9. Згідно документації, дана інструкція використовує тип адресації “Immediate”, займає 2 байти пам’яті та її логіка полягає у тому, аби прочитати байт даних за певним алгоритмом в залежності від типу адресації, після чого помістити його у процесорний регістр A (Accumulator). У даному випадку прочитати потрібно байт, що йде одразу після коду операції. Процесорні прапорці, які дана інструкція може

модифікувати – це “Zero Flag”, біт якого спроставляється у 1 за умови, якщо після операції регістр A = 0, та “Negative Flag”, який проставляється тоді, коли 7 біт регістру A = 1. Передаючи результат у наперед заготовлену фуннкцію – у нас є можливість модифікувати декілька прапорців, використовуючи лише один виклик оновлення прапорців. Такоє варто пам’ятати про те, скільки процесорних циклів займає кожна з таких операцій, і після виконання основної логіки додавати їх до загального лічильника циклів. Під час читання коду операції за допомогою функції “ReadNextByte”, що знаходиться у класі MemoryBus.cs – програмний лічильник автоматично збільшується на 1, оскільки це притаманно для будь-якої інструкції.

```
94 usages
public void UpdateStatusFlags(byte register)
{
    AssignFlag( mask: StatusFlagsConstants.ZERO,
        condition: register == 0);
    AssignFlag( mask: StatusFlagsConstants.NEGATIVE,
        condition: (register & StatusFlagsConstants.NEGATIVE) != 0);
}

49 usages
private void AssignFlag(byte mask, bool condition)
{
    if (condition)
        StatusFlags |= mask;
    else
        StatusFlags &= (byte)~mask;
}
```

Рисунок 2.9 – Функції для оновлення процесорних прапорців

Всього архітектура MOS 6502 нараховує 56 унікальних інструкцій з різними типами адресації, проте описувати їх усіх у рамках даної кваліфікаційної роботи не виглядає раціональним рішенням. Натомість, можливо розбити їх на певні групи, які схожі за своєю логікою та описати їх в цілому. Отже, можна виділити наступні групи:

- Операції збереження та завантаження даних. Прикладами таких інструкцій є LDA (завантаження даних у акумулятор), LDX (завантаження даних у регістр X), STX (збереження даних регістру X у пам’ять).

- Операції передачі між регістрами. До них можна віднести TAX (Передає значення з акумулятора в регістр X), TXA (передає значення з регістра X в акумулятор) та інші.
- Операції для роботи зі стеком. Як приклад можна обрати PHA (кладе значення з акумулятора в стек) та PHP (поміщає значення регістра статусу процесора в стек).
- Логічні операції, серед яких можна виділити CMP (порівнює регістр акумулятору та операнд), AND (виконує побітове "I" між акумулятором і операндом).
- Операції, що включають роботу з арифметикою. Сюди входять ADC (додає значення операнда до акумулятора разом із бітом переносу, що знаходиться у процесорному регістрі “Status Flags”), SBC (віднімає значення операнда до акумулятора разом із бітом переносу).
- Операції обертання та зсуву. Серед них можна виділити ASL та LSR (арифметичний зсув вправо та вліво), ROL та ROR (обертання вправо та вліво з врахуванням біту переносу).
- Операції викликів та переходу. До них належать JMP (перехід до адреси), JSR (перехід до підпрограми) та RTI (вихід з підпрограми). Варто зазначити, що дві останніх інструкції працюють безпосередньо зі стеком і зберігають той адрес, з якого виконувався виклик.
- Операції бранчування (гілки). Дані операції дозволяють за певних умов переходити до різних частин коду. Серед них можна виділити BNE (перехід, якщо нульовий біт регістру “StatusFlags” очищений), BCS (перехід, якщо біт переносу встановлений).
- Операції з регістром “Status Flags”. Дані операції встановлюють та очищають певні прапорці в залежності від інструкції. Наприклад, SEC (встановлює біт переносу) та вже згаданий SEI (вмикає заборону переривань) [9].

Як вже було згадано вище – інструкції реалізовувалися у такому порядку, у якому вони були записані у бінарному файлі гри. Такий підхід дозволяє бачити

прогрес під час розробки та дає можливість займатися відладкою програмного продукту у спрощеному вигляді, оскільки робота йде лише над тим, що безпосередньо використовується під час гри.

Отже, фінальна діаграма сутності, яка репрезентує центральний процесор, виглядає наступним чином:

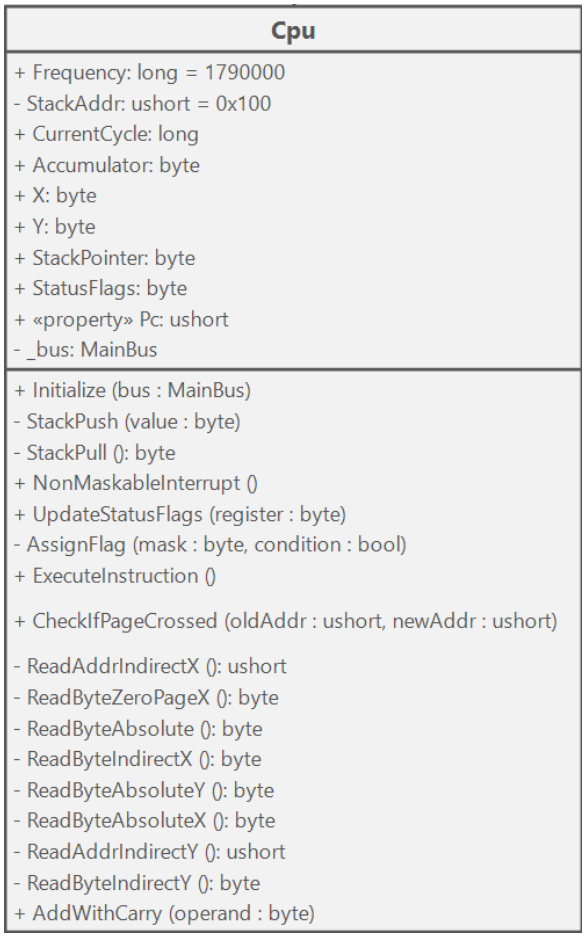

Рисунок 2.10 – Функції для оновлення процесорних прапорців

2.4 Реалізація функціоналу графічного процесору.

Однією з найважливіших складових NES є генератор зображень, який у системі NES називають PPU (Picture Processing Unit). На час виходу ігрової приставки цей компонент вважався доволі просунутим рішенням для роботи з графікою і у рамках кваліфікаційної роботи буде відтворено більшу частину його

функціоналу, що дозволить грати у велику кількість ігор. Для початку потрібно зрозуміти, як він працює в цілому та що собою представляє. У першому розділі було описано його базові характеристики без детального алгоритму роботи, проте вони є доволі важливими, тому варто їх пригадати. Генератор зображень представляє собою доволі складний компонент системи, що функціонує окремо від інших, проте вони завжди синхронізовані. У нього є власні регістри та пам’ять, у які може писати центральний процесор, використовуючи відповідні порти на адресній шині. Оскільки консоль була створена у роки, коли телевізори з технологією електронно-променевої трубки були доволі популярними – всі таймінги роботи PPU залежать безпосереднього від цього механізму і від того, як такі телевізори відмальовували зображення на екрані. Променева трубка випромінювала електрони, рухаючись зліва направо, що дозволяло формувати на екрані горизонтальні лінії, які отримали назву сканлайни (Scanline) [7]. Коли відмальовка горизонтальної лінії завершувалася – променева трубка вимикалася, і поверталася до лівого краю, після чого спускалася нижче. Цей процес отримав назву “Horizontal Blank”, або горизонтальна синхронізація. У свою чергу, коли екран завершує відмальовку останньої лінії – він повинен повернутися на початкову позицію, аби розпочати цей процес спочатку. Для цього був введений інший термін, що отримав назву “Vertical Blank”, або ж вертикальна синхронізація. Ці елементи є критично важливими аби все працювало так, як потрібно, оскільки від цього напряму залежить те, як буде функціонувати PPU.

Отже, варто розпочати написання функціоналу зі створення класу Ppu.cs, у якому буде описуватися вся необіхда логіка. Для початку потрібно додати константи, що являють собою таймінги початку вертикальної синхронізації та довжини кадру у вигляді локальних циклів PPU. Це доволі важливий крок, оскільки він напряму впливає на те, як будуть синхронізуватися графічний та центральний процесори. Також необхідно створити функцію, яка буде виконувати певну роботу із графікою до того моменту, поки не наздожене центральний процесор. У попередніх підрозділах вже було описано те, як відносно частоти оновлення вікна рахується кількість циклів, яку повинен пропрацювати процесор у рамках одного

кадру. Отож, оновлюючи стан центрального процесору – потрібно також оновлювати стан PPU у моменти, коли він знаходиться у режимі вертикальної синхронізації. Для цього, у клас Ppu.cs потрібно додати поле “ppuStatusVBlank” типу bool, яке приймає значення true тоді, коли починається вертикальна синхронізація, а саме напочатку 240 сканлайну. Тепер потрібно створити функцію, яку вирішено назвати “CatchUpToCpuCycle”, що приймає у якості аргумента число, яке є локальним процесорним циклом. Не варто забувати, що PPU рівно у 3 рази швидший за CPU, тому для сихнронізації його потрібно ділити на це число. Суть цього методу полягає у тому, аби всередині, за допомогою циклу while, він працював рівно до того моменту, доки його цикл не почне збігатися з процесорним. Викликатися цей метод повинен у головному класі Emulator.cs у тій ж функції, де і оновлюється процесорний цикл. Варто сказати, що функція “CatchUpToCpuCycle” оновлює цикли, на які орієнтується процесор під час синхронізації обох компонентів. Тобто, напочатку вертикальної синхронізації викликається ще одна створен функція із назвою “CalculateNextVBlankCycle”, яка використовує згадані вище константи для того, аби прорахувати, коли вона відбудеться знову.

```
1 usage
private void CalculateNextVBlankCycle()
{
    NextVBlankCycle += FramePeriod;
}
```

Рисунок 2.11 – Оновлення наступного циклу вертикальної синхронізації.

Реалізований функціонал показано на рисунку 2.11. Також на ньому можна спостерігати, що під час вертикальної синхронізації стається процесорне переривання. Це зроблено для того, аби процесор міг знати момент, у який може розпочати оновлювати графіку, оскільки у цей час PPU не займається відмальовкою і це не викличе ніяких графічних помилок та артефактів.

```
1 usage
public void Update(double dt)
{
    var targetCycle :long  = _cpu.CurrentCycle + (int)(Cpu.Frequency * dt);
    while (_cpu.CurrentCycle < targetCycle)
    {
        if (_cpu.CurrentCycle > VBlankCycle)
        {
            Ppu.CatchUpToCpuCycle(_cpu.CurrentCycle);
            if ((Ppu.PpuRegisters[PpuRegisterConstants.PPUCTRL] & 0x80) != 0)
            {
                _cpu.NonMaskableInterrupt();
            }
        }
        _cpu.ExecuteInstruction();
    }
    Ppu.CatchUpToCpuCycle(_cpu.CurrentCycle);
}
```

Рисунок 2.12 – Сихнронізація компонентів системи

Також було створено діаграму послідовностей, яка показує, яким чином виконується код та як відбувається синхронізація.

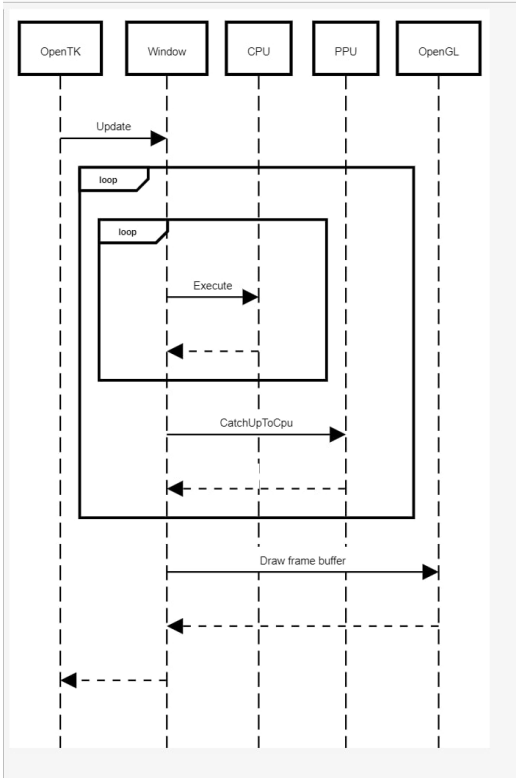

Рисунок 2.13 – Загальна діаграма послідовностей синхронізації

Графічні елементи у NES поділяються на 2 типи: фони та спрайти. Важливо знати те, що у генератору зображень є власна відеопам’ять для кожного з цих типів.

Для фонів вона складає 2 КБ, для спрайтів – 256 байт. У рамках PPU, 2 КБ пам’яті для фонів називаються “Таблицями імен (Nametable)”. Всього таких таблиць є 4, проте фізично використовуватися може лише дві по 1024 байти. Самі графічні елементи знаходяться у вже згаданій раніше частині бінарного файлу гри “CHR ROM”. Ця частина пам’яті складається із двох елементів розміром 4 КБ кожен, які представляють собою “Таблиці шаблонів (Pattern table)”. Усього таких таблиць існує дві, і PPU має прямий доступ до них [7]. На рисунку 2.14 можна бачити, що як ігрові спрайти, так і елементи фону збережені безпосередньо у цих таблицях.

Рисунок 2.14 – таблиці шаблонів у грі Battle City

Таблиця шаблонів являє собою великий блок розміром 32 на 30 тайлів, що у свою чергу є елементами розміром 8 на 8 пікселів та є окремими графічними одиницями [10]. Кожен тайл у таблиці шаблонів займає 16 байт пам’яті, а не 8, як могло б здаватися. Це необхідно для того, аби мати можливість використовувати кольори, про що буде пояснено згодом. В залежності від обраного мапперу – місцезнаходження графічних даних може відрізнятись, проте це не є об’єктом розгляду у даній роботі, оскільки ми розглядаємо саме маппер з індексом 0. Через те, що помістити інформацію про кожен піксель на екрані у такий об’єм пам’яті фізично неможливо – використовується популярний колись режим рендерингу під назвою “Text Mode”. Саме тут потрібно зрозуміти, для чого використовуються таблиці імен та яким чином вони взаємодіють із таблицями шаблонів. По суті те, що знаходиться у відеопам’яті PPU (у таблиці імен) – є вказівником на символ у

таблиці шаблонів, який буде відмальовуватися під час створення кадру. Таблиці шаблонів є статичними, і вони завжди залишаються такими, якими були у бінарному файлі гри, проте таблиця імен постійно змінюється центральним процесором, який може вказувати таким чином, що саме потрібно відмалювати на екрані. Робиться це через адресну шину за допомогою адресів $2000 – $2007. Серед них можна виділити наступні:

- $2000 (PPUCTRL). Цей регістр доступний для запису та контролює різні прапорці, що стосуються функціонування PPU. Він дозволяє зрозуміти, яку таблицю імен потрібно використовувати для відмальовки фонів, яка таблиця патернів використовується для спрайтів та багато іншого.
- $2001 (PPUMASK). Даний регістр використовується для керування рендерингом спрайтів та фону. Зміна певних його значень може вимкнути певні компоненти під час відмальовки.
- $2002 (PPUSTATUS). Регістр статусу відображає стан певних функцій у роботі генератору зображень. Наприклад, біт 7 вказує на те, чи перебуває PPU у стані Vertical Blank (тобто, чи закінчилася відмальовка кадру). Зазвичай процесор очікує цього моменту перед тим, як продовжити свою роботу.
- $2004 (OAMDATA). Цей регістр доступний як на запис, так і на читання. Він дозволяє маніпулювати пам’яттю спрайтів та читати її.
- $2005 (PPUSCROLL). Він відповідає за встановлення позиції прокручування екрану для PPU. Цей регістр використовується для горизонтального та вертикального прокручування ігрового поля: спочатку задається горизонтальне зміщення, а потім вертикальне.
- $2006 (PPUADDR). Оскільки процесор не має можливості писати безпосередньо у відеопам’ять PPU – за допомогою даного регістру він може передати адрес, на який вставониться адрес відеопам’яті. Запис проходить двічі, оскільки оперувати є можливість лише одний байтом даних за раз. Варто зазначити, що даний регістр оперує іншими внутрішніми регістрами, а саме W, V та T. Регістр T у даному контексті

використовується як тимчасовий і зберігає у собі поточну адресу відеопам’яті до того моменту, поки вона не буде записана у регістр V. У реалізованому емуляторі регістр W представляє собою поле типу bool та використовується для того, аби правильно записати адрес, враховуючи молодший та старший байти.

- $2007 (PPUDATA). Цей регістр дозволяє записати дані у відеопам’ять за адресою, що була записана через регістр PPUADDR. Він імплементований так, що кожен запис до нього інкрементує регістр на певне значення, яке залежить від даних регістру PPUCTRL. Це зроблено через те, що кожна операція звернення до пам’яті займає певну кількість циклів, і на такому великому об’ємі інформації це було б нераціональним використанням ресурсів.
- $4014 (OAMDMA). Даний регістр використовується для того, аби процесор міг завантажити інформацію про спрайти у OAM буфер графічного процесору. Використання цього регістру зменшує затримки при оновлюванні спрайтів та дозволяє за один раз повністю оновити вміст даного буферу. На це процесор витрачає 513 або 514 циклів [7].

```
case PpuRegisterConstants.PPUADDR:
{
    if (!_w)
    {
        T &= 0x00FF;
        T = (ushort)(((data << 8) | T) & 0x3FFF);
    }
    else
    {
        T &= 0xFF00;
        T |= data;
        _v = T;
    }
    _w = !_w;
    break;
}
```

Рисунок 2.15 Запис адреси відеопам’яті у внутрішній регістр PPU

Оскільки генератор зображень може оперувати даними – у нього є свій власний адресний простір, що має можливість адресувати 16 КБ пам’яті. Виглядає він наступним чином:

- $0000–$0FFF вказують на першу таблицю шаблонів
- $1000–$1FFF вказують на другу таблицю шаблонів.
- 4 блоки по 1 КБ у діапазоні $2000–$2FFF вказують на таблиці імен, від 1 по 4 відповідно.
- $3F00–$3F1F вказують на дані палітри кольорів, що використовується генератором зображень. Загальний розмір – 32 байти [7].

Після розуміння того, як працює логіка синхронізації та якими є основні компоненти PPU – можна приступати до процесу рендерингу. В цілому, цей процес відбувається наступним чином:

- Визначення сканлайнів, на яких безпосередньо відбувається рендеринг. Це відбувається у рядках з 0 по 240;
- Відмальовка фону, що включає у себе пошук даних у таблиці шаблонів за допомогою відеопам’яті (таблиці імен) та вибір палітри кольорів;
- Відмальовка спрайтів, що включає у себе фільтрування видимих на поточному сканлайні спрайтів, їх рендеринг та вибір кольорів.
- Лінійний запис інформації про пікселі у створений буфер;
- Сповіщення про те, що процес рендерингу завершився [7].

У створеній раніше функції “CatchUpToCpuCycle” класу Ppu.cs потрібно реалізувати цикл, що буде проходитися по уявному екрану із заданим розширенням зліва направо, зверху вниз, починаючи з позиції (0; 0) та закінчуючи позицією (0; 240), оскільки на ній процес рендерингу завершується, і потрібно сповістити про те, що відмальовку кадру у буфер завершено. Найоптимальнішим варіантом рендерингу буде створення масиву даних розміром 256*240, що еквівалентне кількості пікселів на екрані, помножене на 4, оскільки кожен піксель складається із 3 кольорів та альфа компоненту. У свою чергу, процес рендерингу розпочинається з відмальовки фону. Процесор через внутрішні регістри записує у відеопам’ять індекси, які використовуються для вибору тайлів із таблиці шаблонів. Раніше згадувалося, що один тайл розміром 8 на 8 пікселів займає 16 байт пам’яті, оскільки це дозволяє використовувати палітри кольорів. Сам тайл складається з двох площин, які в результаті накладаються одна на одну, в результаті чого на кожен

піксель припадає 2-бітне число від 0 до 3, що дозволяє обрати один із чотирьох доступних кольорів у обраній системній палітрі. Логіка читання цих площин зображена на рисунку 2.16

```
var bkgPatternTableAddr:int = ((PpuRegisters[PpuRegisterConstants.PPUCTRL] >> 4) & 0x1) << 12;
var patternTablePart:int = bkgPatternTableIndex << 4;
var addr = (ushort)(fineY | patternTablePart | bkgPatternTableAddr);
var addrSecond = (ushort)(fineY | patternTablePart | 0x8 | bkgPatternTableAddr);

var firstPlane:byte = ReadVideoMemory(addr);
var secondPlane:byte = ReadVideoMemory(addrSecond);
```

Рисунок 2.16 – Вибір таблиці шаблонів та читання даних тайлу

Оскільки відмальовувати зображення було вирішено по 8 пікселів за 1 раз у рамках циклу – логіка визначення інформації про піксель буде виглядати так, як зображено на рисунку 2.17:

```
//Draw 8 pixels at the same time
for (var i = 0; i < 8; i++)
{
    //Can't be > 0x3
    var bitOnFirstPlane:int = ((1 << (7 - i)) & firstPlane) >> (7 - i);
    var bitOnSecondPlane:int = ((1 << (7 - i)) & secondPlane) >> (7 - i);
    var backgroundTileData:int = (bitOnSecondPlane << 1) | bitOnFirstPlane;
}
```

Рисунок 2.17 – Визначення інформації на основі прощин у конкретному пікселі

Після читання бітів інформації у конкретному пікселі – їх потрібно скласти у 2-бітне число за допомогою відповідних операцій, яке дозволить визначити те, який колір з палітри використовує даний піксель. Сама палітра NES включає у себе 64 кольори, і розділена вона на 16 окремих наборів по 4 кольори [7]. Виникає питання – як дізнатися, який саме набір потрібно обирати? Відповідь на це питання можна отримати, якщо розібратися у ще одному важливому компоненті системи, що має назву “Таблиця атрибутів (Attribute table)”.

Таблиця атрибутів – це компонент, що визначає, які палітри кольорів для фону можуть використовуватися у певних зонах екрану, оскільки вона поділяє його на блоки розмірами по 4 тайли кожен. Іншими словами – один такий блок займає у пам’яті 1 байт і репрезентує простір 32 на 32 пікселів, що поділяється на 4 зони, на кожну з яких виділено 2 біти інформації. Це означало, що гра могла використовувати лише 4 набори по 4 кольори і те, який з них буде обрано, визначалось цими двома бітами у кожному із блоків [10]. Варто зазначити, що таблиця атрибутів використовується лише у парі з ігровим фоном. Тобто, колір та рендеринг спрайтів відбувається окремо від цього процесу і ніяк не залежить від нього. Знаходиться таблиця атрибутів у відеопам’яті, а саме у її останніх 64-ох байтах. Їх потрібно розділити на згадані вище зони та визначити, де саме знаходиться поточний піксель, над яким працює PPU. Також для цього необхідно визначити всі потрібні константи для обрахунків. На рисунку 2.18 продемонстровано процес пошуку необхідного байту у таблиці атрибутів.

```
var pixelIndex:int = (ScreenY * Width + ScreenX) * 4;

var yBlockIndex:int = ScreenY / AttributeTableBlockSize;
var xBlockIndex:int = ScreenX / AttributeTableBlockSize;

var blockXOffset:int = ScreenX - xBlockIndex * AttributeTableBlockSize;
var blockYOffset:int = ScreenY - yBlockIndex * AttributeTableBlockSize;

var attrTableTileAddr:int = attrTableBaseAddr + 8 * yBlockIndex + xBlockIndex;
var attrTableTileByte = ReadVideoMemory((ushort) attrTableTileAddr);
```

Рисунок 2.18 – Пошук потрібного байту у таблиці атрибутів

Можна побачити, що спочатку шукається блок, на якому знаходиться поточний піксель, над яким працює PPU, після чого шукається зсув відносно цього блоку, аби визначити, на якому місці у ньому ми знаходимося. Оскільки один такий блок має розмір 32 на 32 пікселів – стає можливим порахувати, у якій частині блоку ми знаходимося завдяки тому, що розмір блоків прив’язаний то розміру тайлів. На

рисунку 2.19 зображено, яким чином можливо порахувати 2 біти, які використовуються у відповідному блоці

```
var blockQuadrant = 0;

if (blockXOffset < TileSize * 2 && blockYOffset < TileSize * 2)
    blockQuadrant = attrTableTileByte;

else if (blockXOffset >= TileSize * 2 && blockYOffset >= TileSize * 2)
    blockQuadrant = attrTableTileByte >> 6;

else if (blockXOffset < TileSize * 2 && blockYOffset >= TileSize * 2)
    blockQuadrant = attrTableTileByte >> 4;

else if (blockXOffset >= TileSize * 2 && blockYOffset < TileSize * 2)
    blockQuadrant = attrTableTileByte >> 2;
```

Рисунок 2.19 – Обрахунок бітів у поточному блоці

Тепер необхідно реалізувати клас, який буде представляти собою системну палітру кольорів та зможе віддавати необхідні кольори за індексом. Він отримав назву “SystemPalette.cs”, та його логіка зводиться то вивантаження у власну пам’ять інформації про кольори з бінарного файлу. Також він має певні функції, які є необхіднми для роботи з палітрою. Це дозволить гнучно налаштовувати кольори та у будь-який момент завантажити зовсім іншу палітру без необхідності вносити додаткові зміни у код. Був реалізований метод “GetColorByIndex”, що повертає сутність типу “Color”, яка містить собі всю необхідну інформацію про кольори.

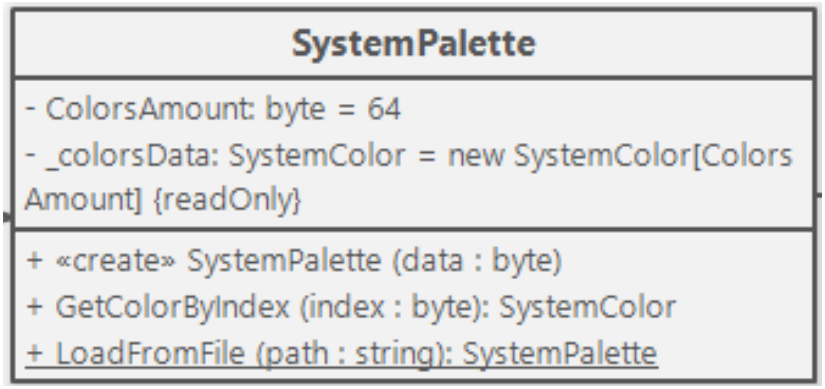

Рисунок 2.20 – Діаграма класу SystemPalette.cs

Отже, змінна “blockQuadrant” вказує на те, яку палітру кольорів потрібно використовувати під час рендерингу поточного пікселя. Враховуючи цю інформацію – тепер можливо записати дані у буфер, що складається з пікселів, які відмальовуються PPU.

```
var bitOnFirstPlane:int = ((1 << (7 - i)) & firstPlane) >> (7 - i);
var bitOnSecondPlane:int = ((1 << (7 - i)) & secondPlane) >> (7 - i);
var backgroundTileData:int = (bitOnSecondPlane << 1) | bitOnFirstPlane;
var backgroundPaletteIndex:byte = GetBackgroundPaletteColorIndex
((byte)(blockQuadrant & 0x3), (byte) backgroundTileData);

var backgroundColor = _systemPalette.GetColorByIndex(backgroundPaletteIndex);

_pixels[pixelIndex] = backgroundColor.R;
_pixels[pixelIndex + 1] = backgroundColor.G;
_pixels[pixelIndex + 2] = backgroundColor.B;
_pixels[pixelIndex + 3] = 255;
```

Рисунок 2.21 – Запис прочитаного кольору у буфер даних

Після того, як інформацію про фон на поточному пікселі було записано – можна перейти до наступного процесу, який відповідає за відмальовку спрайтів. Як було сказано раніше – інформація про спрайти зберігається у окремій частині пам’яті, та заповнюється центральним процесором окремо через спеціальний порт на адресній шині ($4014). Всього пам’яті для спрайтів виділено 256 байт, а максимальна їх кількість – 64. Кожна сутність, що визначає спрайт, складається з наступних компонентів: X координата, Y координата, індекс тайлу та артрибути (палітра, пріоритет, відзеркалення по горизонталі/вертикалі). Перед початком кожного сканлайну, який залучений під час процесу рендеригну – PPU проводить процедуру, що називається “Sprite Evaluation”. Цей процес використовує внутрішню пам’ять графічного процесору, та записує туди ті спрайти, що можуть бути відмальовані на даному сканлайні. Ця внутрішня пам’ять має назву “Internal OAM” та може містити до 8 спрайтів. Якщо ж на лінії відмальвоки було знайдено більше спрайтів – вони просто ігноруються. На оригінальній консолі NES це могло приводити до різного типу артефактів, коли спрайти з’являлися або зникали під час процесу рендерингу. Також варто врахувати те, що цей буфер даних пофинен

очищатися перед початком відмальовки, аби у ньому не були застарілі дані, що могло б створити візуальні проблеми [7].

Ще однією важливою характеристикою спрайтів є те, що вони можуть мати різний розмір. Зазвичай він становить 8 на 8 пікселів, проте у нашому випадку цей розмір становить 8 на 16 пікселів. Під час рендерингу ми знаємо, який режим спрайтів використовується, оскільки інформація про це знаходиться у 5 біті регістру PPUCTRL. Якщо він проставлений – використовується режим 8 на 16. Для простішої перевірки такої інформації було стфорено функцію типу byte, що повертає висоту спрайту, перевіряючи відповідний регістр. Також це було зроблено для того, аби не дублювати код, оскільки даний метод викликається декілька разів у різних функціях. Варто зазначити, що позиція спрайтів затримана на один сканлайн, тому вибір спрайтів для наступного сканлайну повинен відпрацювати наприкінці відмальовки поточного [7]. На рисунку 2.22 показано, яким чином відбувається процес пошуку спрайтів.

```
if (ScreenX == 340)
{
    for (var i = 0; i < 32; i++)
    {
        _internalOam[i] = 0xFF;
    }

    var spritesFound = 0;
    for (var i = 0; i < 64; i++)
    {
        //Sprite height is 8*8 or 8*16 pixels;
        //Sprite data is delayed by one scanline;
        var cmp:int = ScreenY - _oam[i * 4];
        if (cmp >= 0 && cmp < GetCurrentSpriteHeight())
        {
            _internalOam[spritesFound * 4] = _oam[i * 4];
            _internalOam[spritesFound * 4 + 1] = _oam[i * 4 + 1];
            _internalOam[spritesFound * 4 + 2] = _oam[i * 4 + 2];
            _internalOam[spritesFound * 4 + 3] = _oam[i * 4 + 3];
            spritesFound++;
        }
        if (spritesFound == 8)
            break;
    }
}
```

Рисунок 2.22 – Процес вибору спрайтів для наступного сканлайну.

Після даного етапу можна читати інформацію, записану у буфер та відмальовувати її одразу після рендерингу фону. Також потрібно враховувати прозорість, чергу відмальовки та розмір спрайтів. Підхід до написання цієї частини коду є аналогічним до реалізації логіки відмальовки фону, проте є свої нюанси, серед яких варто виділити те, що для вибору кольорів використовується байт атрибутів, у якому знаходиться індекс необхідної палітри. Кольори ж обираються таким самим чином за допомогою двох площин, з яких складається тайл.

Наступним етапом є відмальовка вже створеного буферу з даними безпосередньо у вікно програмного додатку. Для цього потрібно реалізувати певний код за допомогою OpenGL [11], та використати функцію “OnRenderFrame”, яка є частиною фреймворку OpenTK [3]. Оскільки графічний процесор рендерить зображення безпосередньо у масив даних – нам потрібно прочитати його як набір елементів, кожен з яких представляє собою один піксель і має розмір 4 байти (RGBA). Після цього потрібно створити текстуру, та на кожному кадрі заповнювати її вмістом буферу, у який рендерив створений клас Ppu.cs. Також були створені необхідні класи для роботи з текстурами та шейдерами. Для текстур задавався параметр маштабування типу “Nearest”, що дозволяє отримати однаково чітку картинку на різних розширеннях екранів. Він є оптимальним, оскільки в цілому зображення не має згладжених граней.

```
protected override void OnRenderFrame(FrameEventArgs args)
{
    base.OnRenderFrame(args);

    if (!_emulator.Ppu.IsNewFrameReady)
        return;

    _shader.Use();
    _renderTexture.FillTextureData(_emulator.Ppu.GetPixelsData());
    _renderTexture.Use(TextureUnit.Texture0);

    GL.DrawElements(mode: PrimitiveType.Triangles, count: _indices.Length,
        DrawElementsType.UnsignedInt, indices: 0);
    SwapBuffers();
}
```

Рисунок 2.23 – Логіка рендерингу текстури з графікою у вікні емулятору

Отже, можна сказати, що на даному етапі написання функціоналу генератору зображень можна вважати завершеним. Багато ігор можуть мати свої особливості, через що у майбутньому може виникнути потреба дописати його логіку.

2.5 Реалізація функціоналу ігрового контролеру.

У якості пристрою, емуляція якого буде реалізована для операцій введення – було обрано стандартний контролер NES. Кожен контролер має вбудований 8-бітний регістр зсуву, що використовується для послідовного виведення інформації щодо натиснутих кнопок. Для зчитування даних ігрового контролеру використовуються адреси $4016 та $4017 для першого та другого контролеру відповідно. У рамках даної роботи буде реалізовано логіку лише першого конторлеру. Отож, для початку потрібно створити клас, який буде відповідати за логіку натискання кнопок, який було названо “StandardController”. Тепер потрібно зрозуміти алгоритм, за яким гра зчитує дані вводу з ігрового контролера. Це робиться за допомогою послідовних команд на читання пам’яті за адресою $4016, де кожне читання представляє собою стан однієї з натиснутих кнопок та повертає 1 біт інформації з регістру контролеру. Кожна кнопка має свою визначену позицію, тому варто створити перечислення, яке буде включати цю інформацію [7].

```
public enum ButtonType : byte
{
    A = 1 << 0,
    B = 1 << 1,
    Select = 1 << 2,
    Start = 1 << 3,
    Up = 1 << 4,
    Down = 1 << 5,
    Left = 1 << 6,
    Right = 1 << 7,
}
```

Рисунок 2.24 – Визначення послідовності кнопок при зчитуванні

Отож, якщо після читання за адресою $4016 повернувся 0 – це означає, що кнопка не натиснута. Аналогічно з одиницею: якщо повернулося 1 – кнопка на даний момент часу є натистуною. Кожне читання з регістру зсуву, що знаходиться у контролері, зсуває його на одну одиницю праворуч, звідси й назва [7].

Наступним кроком потрібно прив’язати логіку натискання кнопок у програмі самого емулятору до запису даних у створений регістр класу “StandardController”. Для цього було вирішено створити функцію “UpdateInputData”, що викликається головним класом Emulator.cs у рамках головного циклу оновлення. Дана функція приймає клас фреймворку OpenTK [3], що має тип “KeyboardState”, який містить у собі інформацію про те, які кнопки на клавіатурі є натиснутими у рамках поточного кадру.

```
1 usage
public void UpdateInput(KeyboardState keyboardState)
{
    _controller.UpdateInputData(keyboardState);
}
```

Рисунок 2.25 – Оновлення стану системи вводу

Для скидання регістру зсуву процесор записує байт даних за адресою $4016, аби його читання відбувалося з самої першої кнопки. У цей момент потрібно проставити інформацію про те, які кнопки були натистуті, використовуючи згаданий вище клас “KeyboardState” з відповідними методами. Для цього треба створити нову функцію “CheckButtons”, що оновить кнопки та запише інформацію у регістр. Тепер необхідно реалізувати логіку зсуву регістру та повернення інформації про натиснуту кнопку. Для цього було створено функцію “ReadNextButton”, яку зображено на рисунку 2.26. Варто зауважити, що для повернення саме одного біту інформації, потрібно використати відповідну маску. Саме згадана функція буде використовуватися для того, аби повертати значення кнопок під час читання за адресою $4016 [7]. Отже, у кінцевому результаті було отримано працюючий контролер, який емулює натискання клавіш.

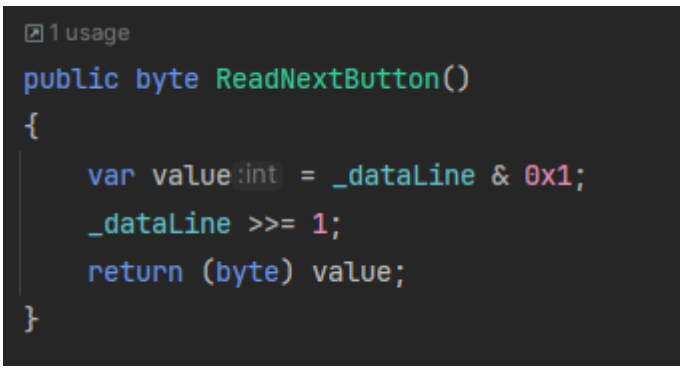

Рисунок 2.26 – Реалізація логіки зсуву на повернення стану кнопки.

РОЗДІЛ 3. ТЕСТУВАННЯ ТА МОЖЛИВІ ПОКРАЩЕННЯ ЕМУЛЯТОРУ

3.1 Тестування емулятору.

Оскільки створення програмного продукту завершується – наступним логічним етапом можна визначити тестування реалізованого функціоналу. Варто зазначити, що емулятор виявився доволі оптимізованим, що дає можливість використовувати його на широкому спектрі різних персональних комп’ютерів. Можна сказати, що продукт тестувався безпосередньо під час розробки, оскільки неправильне написання логіки будь-якого з компонентів привело б до повної недієздатності програмного продукту. Через те, що тестування емулятору являє собою доволі складний процес – розробники, що спеціалізуються на цьому, підійшли до цієї задачі з креативом. Для перевірки роботи центрального процесору на корекстність виконання інструкцій та оперування даними, потрібно завантажити спеціальні бінарні файли та прочитати їх у емуляторі так, наче це звичайні ігрові додатки. Вони виконуються як стандартні програми, проте всередині вони реалізовані так, що поступово тестують кожну відому інструкцію, після чого перевіряють значення регістрів, якими вони маніпулюють та вміст оперативної пам’яті. Такий підхід дозволяє зрозуміти, чи правильно працює центральний процесор, та доволі швидко знайти проблеми у коді, що описує логіку. Також варто сказати, що для перевірки варто використовувати логи, які записані емуляторами, працездатність яких перевірена часом та ретельним тестуванням.

```
C000  4C F5 C5  JMP $C5F5                       A:00 X:00 Y:00 P:24 SP:FD PPU:  0, 21 CYC:7
C5F5  A2 00     LDX #$00                        A:00 X:00 Y:00 P:24 SP:FD PPU:  0, 30 CYC:10
C5F7  86 00     STX $00 = 00                    A:00 X:00 Y:00 P:26 SP:FD PPU:  0, 36 CYC:12
C5F9  86 10     STX $10 = 00                    A:00 X:00 Y:00 P:26 SP:FD PPU:  0, 45 CYC:15
C5FB  86 11     STX $11 = 00                    A:00 X:00 Y:00 P:26 SP:FD PPU:  0, 54 CYC:18
C5FD  20 2D C7  JSR $C72D                       A:00 X:00 Y:00 P:26 SP:FD PPU:  0, 63 CYC:21
C72D  EA        NOP                             A:00 X:00 Y:00 P:26 SP:FB PPU:  0, 81 CYC:27
C72E  38        SEC                             A:00 X:00 Y:00 P:26 SP:FB PPU:  0, 87 CYC:29
C72F  B0 04     BCS $C735                       A:00 X:00 Y:00 P:27 SP:FB PPU:  0, 93 CYC:31
C735  EA        NOP                             A:00 X:00 Y:00 P:27 SP:FB PPU:  0,102 CYC:34
C736  18        CLC                             A:00 X:00 Y:00 P:27 SP:FB PPU:  0,108 CYC:36
C737  B0 03     BCS $C73C                       A:00 X:00 Y:00 P:26 SP:FB PPU:  0,114 CYC:38
C739  4C 40 C7  JMP $C740                       A:00 X:00 Y:00 P:26 SP:FB PPU:  0,120 CYC:40
C740  EA        NOP                             A:00 X:00 Y:00 P:26 SP:FB PPU:  0,129 CYC:43
C741  38        SEC                             A:00 X:00 Y:00 P:26 SP:FB PPU:  0,135 CYC:45
```

Рисунок 3.1 Приклад логування даних емулятору

Як вже було згадано – мінімальна помилка у обрахунках та логіці центрального процесору може привести до критичних помилок та багів під час гри, тому перевірка інструкцій та роботи системи в цілому є критично важливою складовою під час тестування.

Другим важливим етапом тестування інформаційної системи є перевірка того, для чого вона створювалася. Іншими словами, потрібно візуально впевнитися, що все працює справно і картинка показується такою, якою вона повинна бути. Для цього тестування було обрано ігрові додатки, згадані ще у першому розділі. Візуальне тестування можна поділити на наступні етапи:

- Правильність графіки. Для цього потрібно перевірити, чи всі спрайти та фони відображаються правильно. Це включає в себе перевірку відсутності зникання, перекриття або неправильного рендерингу графічних елементів.
- Синхронізація кадрів. Це доволі важливий пункт тестування, оскільки неправильна синхронізація може призвести до "розривів" зображення або миготіння.
- Кольорова палітра. Цей пункт включає в себе коректне відтворення кольорової палітри NES, оскільки неточності можуть вплинути на візуальне сприйняття гри.
- Поведінка фону і спрайтів. Тут в основному перевіряється взаємодія між ними та корекнтість роботи прозорості.

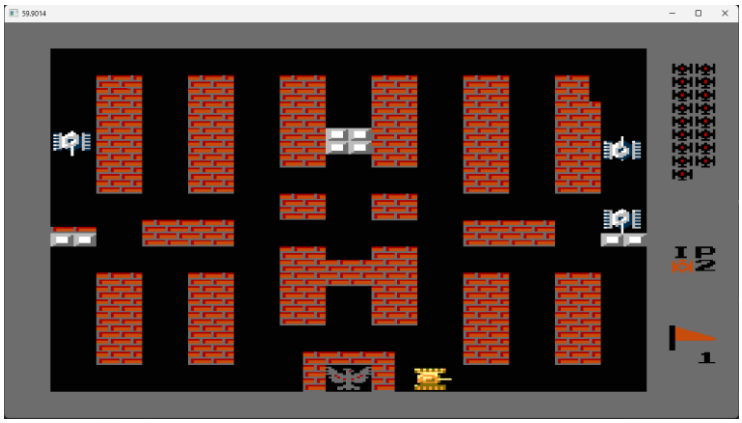

Рисунок 3.2 – Тестування гри Battle City на розробленому емуляторі

Отже, спочатку було протестовано гру Battle City (1985). Це дозволило зрозуміти, що під час гри не виникає проблем з керуванням, інструкціями центрального процесору та системою керування. Як вже було згадано раніше – візуальне тестування дозволяє зрозуміти, чи правильно функціонує процесор та генератор зображень. На рисунку 3.2 показано, що фон та спрайти відмальовуються коректно, і в цілому проблем із грою не виникає. Також за допомогою такого підходу були протестовані компоненти графічного процесору, що відповідають за чергу відмальовки спрайтів та фонів. Для того, аби верифікувати, що гру можливо пройти та емулятор працює справно – на рисунку 3.3 зображено кінець першого рівня зі статистикою.

Рисунок 3.3 – Завершення ігрового рівня

Потрібно зазначити, що вказана гра (Battle City) дозовляє звернути увагу на те, чи правильно взаємодіють між собою різні компоненти в залежності від типів спрайтів, що використовуються у грі, оскільки у даному випадку їхній розмір становить 8x16 пікселів, що додатково дозволяє перевірити правильність реалізації такої логіки зі сторони графічного процесору.

Наступним ігровим додатком, який було обрано у рамках тестування, став Mario Bros (1983). Це доволі популярна гра, яка створена на основі того ж мапперу з індексом 0, і є хорошим варіантом для перевірки працездатності системи.

Рисунок 3.4 – Тестування гри Mario Bros на розробленому емуляторі

На рисунку 3.4 можна побачити, як візуально виглядає гра Mario Bros, протестована на емуляторі. Це дозволяє впевнитися, що вона також працює коректно і графіка відмальовується так, як задумано. Враховуючи все сказане вище – можна сказати, що тестування інструкцій та графічного процесору пройшло успішно.

3.2 Можливі покращення емулятору у майбутньому.

Оскільки реалізація всіх можливостей оригінальної системи NES може зайняти доволі великий проміжок часу – у даній роботі увага була зосереджена здебільшого на її основних компонентах. Функціонал емулятору не обмежується тим, що було імплементовано у даній роботі. Існує велика кількість додаткових речей, що дозволять покращити ігровий досвід гравців та ентузіастів. Серед них можна виділити:

- Реалізація системи аудіо. Це доволі важливий пункт, який безпосередньо впливає на зацікавленість грою. Оскільки це доволі комплексна система – її реалізація може зайняти доволі багато часу.

- Імплементація функціоналу різноманітних мапперів. У даній кваліфікаційній роботі було реалізовано лише маппери з індексом 0 та 2. Хоч вони й дозволяють запускати понад 70% інсуючих під платформу NES ігор – доволі велика доля припадає і на інші маппери, що у свою чергу робить цей пункт також доволі важливим.
- Кросплатформність. Оскільки у сучасному світі існує декілька різних операційних систем – додатки, що орієнтуються на велику цільову авдиторію, повинні мати можливість працювати на більшості з них. Оскільки проєкт був створений на основі фреймворку .Net версії 6 – це завдання не повинно зайняти велику кількість часу. Також варто врахувати те, що графічний фреймворк, що використовується у даній роботі (OpenTK [3]), також має можливість працювати на різних платформах.

Отже, можна сказати, що є ще багато компонентів емулятору, які можуть бути реалізованими або потребують доробки.

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Природне середовище і його забруднення

Природне середовище є надважливою складовою у житті всього живого та складається з атмосфери, вод, ґрунтів, лісів, рослин, тварин та інших живих організмів, що населяють Землю. Воно відіграє критичну роль у підтримці життя на планеті та забезпеченні стабільного функціонування екосистем [13].

Тема забруднення природнього середовища є критично важливою, оскільки вона безпосередньо впливає на якість життя. Варто розуміти, що чинників забрудення навколишьного середовища може бути дуже багато, проте більшість з них викликана людським фактором. Людський фактор (ЛФ) включає фізіологічні, психофізіологічні, антропометричні та професійні особливості, які можуть сприяти виникненню небезпек. Від 40 до 80 % небезпек виникають у результаті діяльності людини [13].

Забрудення навколишнього середовища поділяється на велику кількість факторів, які на це впливають. Варто виділити основні з них:

1. Атмосферне забруднення. Сюди відносяться викиди від транспортних засобів, промислових підприємств, теплових електростанцій, спалювання відходів.
2. Водне забруднення. Сюди відносяться скидання промислових і побутових стічних вод, сільське господарство, нафтові розливи, морські судна тощо.
3. Ґрунтове забруднення. Сюди відносяться використання пестицидів та добрив, промислові відходи, незаконні звалища, аварії на підприємствах.
4. Радіаційне забруднення. Основними причинами такого забруднення є аварії на атомних електростанціях, промислові процеси, видобуток та використання радіоактивних матеріалів [13].

На мою думку, однією з найбільших проблем, яка школить навколишньому середовищу можна назвати те, що технічний процес постійно розвивається і це провокує забруднення природи навколо нас. Він приносить з собою комфорт і процвітання, проте це супроводжується різким збільшенням забруднення навколишнього середовища через накопичення синтетичних хімічних сполук, кількість яких сягнула 400 тисяч, більшість з яких є токсичними. Сучасний рівень електромагнітного фону Землі перевищує природний у 200 000 разів, багато видів тварин і рослин зникли, і з'явилися нові техногенні небезпеки. Основними джерелами забруднення навколишнього середовища є теплові електростанції, автомобільний транспорт, металургійні та хімічні заводи, целюлозно-паперова промисловість і сучасні методи сільського господарства [13].

Також однією з проблем людства та навколишнього середовища можна виділити відходи та сміття. Порушення екологічної рівноваги призводить до зниження родючості ґрунтів, втрати біорізноманіття та загрози для екосистем. Більшість відходів, що потрапляють на сміттєзвалища, можуть бути перероблені або використані повторно, але недостатня увага до цього питання призводить до втрати цінних ресурсів та зростання екологічних проблем. Викиди шкідливих газів, таких як метан, не лише погіршують якість повітря, а й сприяють парниковому ефекту та глобальному потеплінню, що має серйозні наслідки для клімату та екологічної стійкості планети [14].

Враховуючи сказане вище, можна сказати, що важливим є те, як люди будуть відноситися до природи та яким чином будуть запобігати виникненню ситуацій, які будуть негативно на неї впливати. Запобігання надзвичайним ситуаціям включає комплекс заходів, які охоплюють правові, соціально-економічні, політичні, організаційно-технічні, санітарно-гігієнічні та інші аспекти. Ці заходи спрямовані на забезпечення техногенної та природної безпеки, оцінку ризиків, і передбачають своєчасне реагування на потенційні загрози. Використовуючи дані моніторингу, експертизи, досліджень і прогнозів, вдається запобігти перетворенню загроз у реальні надзвичайні ситуації або мінімізувати їх можливі наслідки.

Рисунок 4.1 – Забруднення навколишнього середовища

Отже, навколишнє середовище є критичним для існування людей та підтримки екосистеми нашої планети, оскільки безпосередньо впливає на якісь нашого життя та залежить безпосередньо від нас. Потрібно бережно відноситися до природи та того, що нас оточує. Необережне ставлення до навколишнього середовища та не розуміння того, яким чином можна запобігати забрудненню можуть стати чинниками для виникнення катастрофи.

4.2 Психофізіологічне розвантаження для працівників

Правильний відпочинок та моральне розвантаження є важливими складовими робочого процесу, які безпосередньо впливають на продуктивність працівників та їх психофізіологічний стан. Для визначення на науковій основі методів і шляхів поліпшення та оздоровлення умов праці на виробництві, забезпечення правильного ритму, режиму праці й відпочинку необхідно враховувати вимоги психології стосовно праці людини (вивчення чинників, що впливають на працездатність людини – втома, нервове напруження, монотонність роботи) [15].

Для того, аби зрозуміти, на що потрібно звертати увагу під час роботи над психологічним розвантаженням – треба перечислити причини, які є чинниками такого навантаження: втома, викликана великим фізичним перевантаженням, розумовим перенапруженням, зоровим і слуховим перевантаженням; монотонність праці; стресові та надзвичайні ситуації; хворобливий стан; невідповідність анатомо-фізіологічних і психічних особливостей організму людини виконуваній роботі. Сьогодні ще недостатньо враховуються фізіологічні, психофізіологічні, психологічні особливості людини під час створення складних технічних систем і конструкцій машин, приладів і систем управління [15].

На мою думку, варто виділити вимоги, які є необхідними для підтримки психофізіологічного стану людини на належному рівні:

1. Ергономічні вимоги. Ергономіка вивчає оптимальні умови праці, що сприяють зменшенню фізичних і психічних навантажень. Важливими аспектами є правильна організація робочого місця, вибір робочої пози, розміщення інструментів та обладнання. Ергономічно облаштоване робоче місце дозволяє знизити втому та підвищити продуктивність праці.
2. Психофізіологічні вимоги. У себе вони включають встановлення відповідності між швидкісними, енергетичними, зоровими та іншими фізіологічними можливостями людини в процесі. Також важливо вводити раціональні режими праці та відпочинку. Скорочення обсягу інформації допомагає знизити нервово-емоційне напруження та фізичне навантаження. Професійний відбір є ще одним критичним аспектом [15].
3. Вимоги до метеорологічних умов. Метеорологічні умови виробничих приміщень значною мірою впливають на фізіологічні функції організму працівника. Важливими показниками є температура, вологість, швидкість руху повітря. Оптимальні метеорологічні умови сприяють збереженню здоров'я та підвищенню працездатності працівників. Зокрема, правильна терморегуляція допомагає уникнути перегріву або переохолодження, що позитивно впливає на самопочуття та продуктивність [15].

4. Естетичні вимоги. Колір в технічній естетиці також має значний вплив на психофізіологічний стан працівників. Використання оптимальних кольорових рішень в інтер'єрі виробничих приміщень сприяє створенню комфортної атмосфери, що позитивно впливає на самопочуття та продуктивність праці. Кольори можуть впливати на настрій, рівень стресу та загальний стан працівника [16].
5. Вимоги до виробничого середовища. Вдосконалення технологічного процесу спрямоване на створення комфортної виробничої атмосфери, що знижує негативний вплив на працівників. Оптимізація робочих процесів, зниження рівня шуму та вібрації, поліпшення освітлення є ключовими заходами для забезпечення психофізіологічного розвантаження. Використання сучасних технологій та автоматизація виробничих процесів допомагають зменшити фізичні навантаження та підвищити загальну ефективність праці [16].

Отже, психофізіологічне розвантаження працівників є важливим елементом охорони праці, що спрямоване на збереження здоров'я та підвищення продуктивності праці. Врахування фізичних, психологічних та ергономічних аспектів у організації трудового процесу дозволяє створити оптимальні умови праці, знизити рівень втоми та нервового напруження, що позитивно впливає на загальний стан працівників та результати їхньої діяльності.

ВИСНОВКИ

Розробка платформи для емуляції ігрової приставки NES стала важливим етапом у моєму професійному розвитку як розробника. Цей проєкт об'єднує різноманітні аспекти програмування, включаючи створення графічного інтерфейсу, обробку вхідних даних та ефективне управління системними ресурсами. Використання мови програмування C# дозволило розробити продуктивне та надійне програмне забезпечення, а бібліотека OpenTK забезпечила необхідні інструменти для відтворення графіки та взаємодії з користувачем. Також варто сказати, що застосування системи контролю версій Git у поєднанні з клієнтом SourceTree надало можливість безпечно зберігати та організовувати історію розробки проєкту. Це забезпечило надійний контроль версій та сприяло спільній роботі над кодом, що є важливим для будь-якого масштабного проєкту. Протягом процесу розробки було створено та проаналізовано UML діаграми, які допомогли структурувати ідеї та забезпечити логічну послідовність розробки. Використання UML діаграм дозволило краще зрозуміти взаємодію між компонентами системи та забезпечило чітке уявлення про архітектуру програмного продукту. Особливо це видно на прикладі діаграми послідовностей, що описує синхронізацію системних компонентів.

Серед найважливіших компонентів можна виділити реалізацію логіки центарльного процесору та генератору зображень. Це доволі складні компоненти, які потребували високого рівня підготовленості та знать, проте були реалізовані у рамках даної кваліфікаційної роботи. Також надважливим пунктом є продумування синхронізації цих компонентів, що дозволило отримати справний програмний продукт і чітке та плавне зображення.

Я вважаю, що здобув глибоке розуміння того, як працюють різні компоненти програмного забезпечення, і як вони взаємодіють один з одним. Це дало змогу не лише поліпшити мої навички програмування, але й підвищити загальну компетентність у галузі розробки програмних систем. Також важливою ціллю

проєкту було створення такого емулятору, який гравці та ентузіасти зможуть використовувати для своїх потреб. Було реалізовано два основних маппери, які підтримують більшість ігор платформи. Емулятор було протестовано візуально та перевірено на баги за домогою бінарних файлів для тестування, що дозволило впевнитися у тому, що все працює справно.

Ще одним важливим пунктом, який розглядався у рамках третього розділу, було обговорення майбутніх покращень, оскільки емулятор – це такий програмний продукт, який завжди може отримувати новий функціонал. Серед важливих пунктів цієї теми увага була зосереджена на реалізації нових мапперів та на імплементанцію основної логіки платформи.

Отже, цілі, поставлені на початку кваліфікаційної роботи, можна вважати успішно виконаними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. NET | Build. Test. Deploy [Електронний ресурс] – Режим доступу до ресурсу: https://dotnet.microsoft.com/en-us/.

2. C# language documentation [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/csharp/.

3. OpenTK [Електронний ресурс] – Режим доступу до ресурсу: https://opentk.net/.

4. Rider: The Cross-Platform .NET IDE from JetBrains [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/rider/.

5. Git [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Git/.

6. SourceTree [Електронний ресурс] – Режим доступу до ресурсу: https://www.sourcetreeapp.com/.

7. Nesdev Wiki [Електронний ресурс] – Режим доступу до ресурсу: https://www.nesdev.org/wiki/Nesdev_Wiki/.

8. Mappers Matter [Електронний ресурс] – Режим доступу до ресурсу: https://somethingnerdy.com/mappers-matter/.

9. 6502 Instruction Set [Електронний ресурс] – Режим доступу до ресурсу: https://www.masswerk.at/6502/6502_instruction_set.html/.

10. Event loop [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Event_loop/.

11. An overview of NES rendering [Електронний ресурс] – Режим доступу до ресурсу: https://austinmorlan.com/posts/nes_rendering_overview/.

12. OpenGL - The Industry Standard for High Performance Graphics [Електронний ресурс] – Режим доступу до ресурсу: https://opengl.org/.

13. М. С. Одарченко, А. М. Одарченко, В. І. Степанов, Я. М. Черненко «Основи охорони праці» підручник / М. С. Одарченко, А. М. Одарченко, В. І. Степанов, Я. М. Черненко. – Харків: Стиль-Издат, 2019. – 25 – 31 с.

14. Екологічні проблеми зберігання та утилізації відходів в Україні [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.legalaid.gov.ua/.

15. О. Г. Левченко, О. В. Землянська, Н. А. Праховнік, В. В. Зацарний «Безпека життєдіяльності та цивільний захист» навчальний посібник / О. Г. Левченко, О. В. Землянська, Н. А. Праховнік, В. В. Зацарний. – Київ: КПІ ім. Ігоря Сікорського, 2019. – 12 – 14 с.

16. О. Г. Левченко, О. В. Землянська, Н. А. Праховнік, В. В. Зацарний «Безпека життєдіяльності та цивільний захист» навчальний посібник / О. Г. Левченко, О. В. Землянська, Н. А. Праховнік, В. В. Зацарний. – Київ: КПІ ім. Ігоря Сікорського, 2019. – 110 – 190 с.

ДОДАТКИ

Лістинг 1 – код класу Emulator.cs

```
using Emulator.Constants;
using Emulator.Controller;
using Emulator.Logs;
using OpenTK.Windowing.GraphicsLibraryFramework;

namespace Emulator;

public class Emulator
{
    public static bool IsLoggingEnabled;
    public static Logger Logger;

    public Ppu Ppu;

    private MainBus _bus;
    private Cpu _cpu;
    private Ram _ram;
    private Rom _rom;
    private StandardController _controller;

    //_ppu.NextVBlankCycle + 2 because of the integer rounding.
    long VBlankCycle => (Ppu.NextVBlankCycle + 2) / 3;

    public void Initialize(string romPath, string palettePath)
    {
        var romLoader = new RomLoader();
        _rom = romLoader.Load(romPath);
        _ram = new Ram();

        _controller = new StandardController();

        var palette = SystemPalette.LoadFromFile(palettePath);
        Ppu = new Ppu(_rom, palette);
```

```
        _cpu = new Cpu();
        _bus = new MainBus();
        _cpu.Initialize(_bus);
        _bus.Initialize(_cpu, Ppu, _ram, _rom, _controller);

        ResetCpu();

        Ppu.NextVBlankCycle = Ppu.VblankOffset;
    }

    private void ResetCpu()
    {
        var firstByte = _bus.ReadMemoryByAddr(0xFFFC);
        var secondByte = _bus.ReadMemoryByAddr(0xFFFD);
        _cpu.Pc = (ushort)(firstByte | secondByte << 8);
        _cpu.CurrentCycle += 7;
        _cpu.StatusFlags = 0x24;
    }

    public void UpdateInput(KeyboardState keyboardState)
    {
        _controller.UpdateInputData(keyboardState);
    }

    public void Update(double dt)
    {
        var targetCycle = _cpu.CurrentCycle + (int)(Cpu.Frequency * dt);
        while (_cpu.CurrentCycle < targetCycle)
        {
            if (_cpu.CurrentCycle > VBlankCycle)
            {
                Ppu.CatchUpToCpuCycle(_cpu.CurrentCycle);
                if ((Ppu.PpuRegisters[PpuRegisterConstants.PPUCTRL] & 0x80)
!= 0)
                {
                    _cpu.NonMaskableInterrupt();
                }
            }
            _cpu.ExecuteInstruction();
```

```
        }
        Ppu.CatchUpToCpuCycle(_cpu.CurrentCycle);
    }
}
```

Лістинг 2 – код класу EmulatorWindow.cs

```
using System.Drawing;
using Emulator.Logs;
using Emulator.Shaders;
using OpenTK.Graphics.OpenGL;
using OpenTK.Mathematics;
using OpenTK.Windowing.Common;
using OpenTK.Windowing.Desktop;
using OpenTK.Windowing.GraphicsLibraryFramework;

namespace Emulator;

public class EmulatorWindow : GameWindow
{
    private Shader _shader;
    private bool _romSelected;

    private readonly float[] _vertices =
    {
        -1f, -1f, 0.0f, 0.0f, 1f,
        -1f, 1f, 0.0f, 0.0f, 0f,
        1f, 1f, 0.0f, 1f, 0f,
        1f, -1f, 0.0f, 1f, 1f
    };

    private readonly uint[] _indices = {
        0, 1, 3
        1, 2, 3
    };

    private int _elementBufferObject;
    private int _vertexBufferObject;
```

```
    private int _vertexArrayObject;

    private Texture _renderTexture;
    private Emulator _emulator;

    public EmulatorWindow(GameWindowSettings gameWindowSettings,
NativeWindowSettings nativeWindowSettings) : base(gameWindowSettings,
nativeWindowSettings)
    {
        Size = new Vector2i(1280, 720);
        _emulator = new Emulator();
        _renderTexture = Texture.GenerateTexture(256, 240);
        _shader = new Shader(@"Shaders\Shader.vert",
            @"Shaders\Shader.frag");
        _shader.Use();
    }

    private void InitializeRom(string romName)
    {
        _emulator.Initialize(@$"Resources\{romName}.nes",
@"Resources\palette.pal");
        _romSelected = true;
    }

    protected override void OnUnload()
    {
        _shader.Dispose();
        base.OnUnload();
    }

    protected override void OnLoad()
    {
        base.OnLoad();
        GL.ClearColor(Color.Black);

        _vertexArrayObject = GL.GenVertexArray();
        GL.BindVertexArray(_vertexArrayObject);

        _vertexBufferObject = GL.GenBuffer();
```

```
        GL.BindBuffer(BufferTarget.ArrayBuffer, _vertexBufferObject);
        GL.BufferData(BufferTarget.ArrayBuffer, _vertices.Length *
sizeof(float), _vertices, BufferUsageHint.StaticDraw);

        _elementBufferObject = GL.GenBuffer();
        GL.BindBuffer(BufferTarget.ElementArrayBuffer,
_elementBufferObject);
        GL.BufferData(BufferTarget.ElementArrayBuffer, _indices.Length *
sizeof(uint), _indices, BufferUsageHint.StaticDraw);

        var vertexLocation = GL.GetAttribLocation(_shader.Handle,
"aPosition");
        GL.EnableVertexAttribArray(vertexLocation);
        GL.VertexAttribPointer(vertexLocation, 3,
VertexAttribPointerType.Float, false, 5 * sizeof(float), 0);

        var texCoordLocation =
GL.GetAttribLocation(_shader.Handle,"aTexCoord");
        GL.EnableVertexAttribArray(texCoordLocation);
        GL.VertexAttribPointer(texCoordLocation, 2,
VertexAttribPointerType.Float, false, 5 * sizeof(float), 3 *
sizeof(float));
    }

    protected override void OnResize(ResizeEventArgs e)
    {
        base.OnResize(e);
        GL.Viewport(0, 0, e.Width, e.Height);
    }

    protected override void OnRenderFrame(FrameEventArgs args)
    {
        base.OnRenderFrame(args);

        if (!_romSelected)
            return;

        if (!_emulator.Ppu.IsNewFrameReady)
            return;
```

```
    _shader.Use();
    _renderTexture.FillTextureData(_emulator.Ppu.GetPixelsData());
    _renderTexture.Use(TextureUnit.Texture0);

    GL.DrawElements(PrimitiveType.Triangles, _indices.Length,
        DrawElementsType.UnsignedInt, 0);
    SwapBuffers();
}

protected override void OnUpdateFrame(FrameEventArgs args)
{
    base.OnUpdateFrame(args);

    if (KeyboardState.IsKeyPressed(Keys.F1))
    {
        Emulator.Logger?.Dispose();
        var shouldLog = !Emulator.IsLoggingEnabled;
        if (shouldLog)
            Emulator.Logger = new Logger();

        Emulator.IsLoggingEnabled = shouldLog;
    }

    Title = (1 / args.Time).ToString("F4");
    if (Emulator.IsLoggingEnabled)
        Title += " [Recording]";

    if (!_romSelected)
    {
        if (KeyboardState.IsKeyPressed(Keys.D1))
            InitializeRom("contra");
        if (KeyboardState.IsKeyPressed(Keys.D2))
            InitializeRom("rom");
        if (KeyboardState.IsKeyPressed(Keys.D3))
            InitializeRom("bike");
        if (KeyboardState.IsKeyPressed(Keys.D4))
            InitializeRom("duck");
        if (KeyboardState.IsKeyPressed(Keys.D5))
```

```
                InitializeRom("kong");
            if (KeyboardState.IsKeyPressed(Keys.D6))
                InitializeRom("mario");
            if (KeyboardState.IsKeyPressed(Keys.D7))
                InitializeRom("ducktales");
            return;
        }
        _emulator.UpdateInput(KeyboardState);
        _emulator.Update(args.Time);
    }
}
```

Лістинг 3 – код класу MainBus.cs

```
using Emulator.Constants;
using Emulator.Controller;

namespace Emulator;

public class MainBus
{
    private Cpu _cpu;
    private Ppu _ppu;
    private Ram _ram;
    public Rom _rom;

    private StandardController _controller;
    public void Initialize(Cpu cpu, Ppu ppu, Ram ram, Rom rom,
StandardController controller)
    {
        _cpu = cpu;
        _ppu = ppu;
        _ram = ram;
        _rom = rom;
        _controller = controller;
    }

    public byte ReadNextByte()
```

```
{
    var data = ReadMemoryByAddr(_cpu.Pc);
    _cpu.Pc += 1;
    return data;
}

public ushort ReadAddrAbsolute()
{
    return (ushort)(ReadNextByte() | ReadNextByte() << 8);
}

//16-bit addressing mode.
public byte ReadMemoryByAddr(ushort address)
{
    switch (address)
    {
        case <= 0x1FFF:
            return _ram.Read(address);
        case <= 0x2007:
        {
            _ppu.CatchUpToCpuCycle(_cpu.CurrentCycle);
            return _ppu.ReadRegister((byte)(address - 0x2000));
        }
        case 0x4016:
            return (byte) (_controller.ReadNextButton() + 0x40);
        case >= 0x8000:
            return _rom.ReadPrg((ushort) (address - 0x8000));
    }
    return 0;
}

public void WriteMemory(ushort address, byte data)
{
    switch (address)
    {
        //Stack
        case >= 0x0100 and <= 0x01FF:
            _ram.Write(address, data);
            break;
```

```
            //Ram
            case <= 0x07FF:
                _ram.Write(address, data);
                break;
            //PPU
            case >= 0x2000 and <= 0x2007:
            {
                _ppu.CatchUpToCpuCycle(_cpu.CurrentCycle);
                _ppu.WriteRegister((byte)(address - 0x2000), data);
                break;
            }
            //OAM DMA
            case 0x4014:
            {
                for (var i = 0; i < 256; i++)
                {
                    var addr = (ushort)((data << 8) | i);
                    var spriteData = ReadMemoryByAddr(addr);
                    _ppu.WriteRegister(PpuRegisterConstants.OAMDATA,
spriteData);
                }
                _cpu.CurrentCycle += 513;
                break;
            }
            //APU
            case >= 0x4000 and <= 0x4013:
            case 0x4015:
            case 0x4017:
            {
                break;
            }
            //Controller
            case 0x4016:
            {
                _controller.CheckButtons(data == 0);
                break;
            }
            case >= 0x8000:
            {
```

```
                _rom.WriteMemory(address, data);
                break;
            }
            default:
            {
                Console.WriteLine($"Invalid write addr: {address:X4}");
                break;
            }
        }
    }
}
```

Лістинг 4 – код класу Rom.cs

```
using Emulator.Enums.INES;
using Emulator.Mappers;

namespace Emulator;

public class Rom
{
    public const ushort PRG_BANK_SIZE = 0x4000;
    public const ushort CHR_BANK_SIZE = 0x2000;

    private bool _useChrRam;

    public MirroringType RomMirroringType;

    public IMapper Mapper { get; }

    public byte ReadPrg(ushort addr)
    {
        return Mapper.ReadPrg(addr);
    }

    public void WriteMemory(ushort addr, byte data)
    {
        switch (addr)
```

```
        {
            case >= 0x8000:
            {
                Mapper.SelectedBank = (byte) (data & 0x7);
                break;
            }
        }
    }

    public void WriteChrRam(ushort addr, byte data)
    {
        Mapper.WriteChrRam((ushort) (addr & (CHR_BANK_SIZE - 1)), data);
    }

    public byte ReadChr(ushort addr)
    {
        if (!_useChrRam)
            return Mapper.ReadChrRom(addr);

        return Mapper.ReadChrRam(addr);
    }

    public Rom(byte[] prg, byte[] chr, byte romFlags6, byte romFlags7, byte
prgSize, byte chrSize)
    {
        var prgMask = PRG_BANK_SIZE * prgSize - 1;
        var chrMask = CHR_BANK_SIZE * chrSize - 1;
        _useChrRam = chrSize == 0;

        var mapperId = (byte)(((romFlags6 & 0xF0) >> 4) | ((romFlags7 &
0xF) << 4));
        Mapper = MapperFactory.CreateById(mapperId);
        Mapper.Init(prg, chr, prgMask, chrMask);

        RomMirroringType = (byte)(romFlags6 & 0x1) == 0 ?
MirroringType.Horizontal : MirroringType.Vertical;
    }
}
```

Лістинг 5 – код класу SystemPalette.cs

```
namespace Emulator;

public class SystemPalette
{
    private const byte ColorsAmount = 64;
    private readonly SystemColor[] _colorsData = new
SystemColor[ColorsAmount];

    public SystemPalette(byte[] data)
    {
        var arrayIndex = -1;
        for (var i = 0; i < 64; i++)
        {
            var r = data[++arrayIndex];
            var g = data[++arrayIndex];
            var b = data[++arrayIndex];
            _colorsData[i] = new SystemColor(r, g, b);
        }
    }

    public SystemColor GetColorByIndex(byte index)
    {
        return _colorsData[index];
    }

    public static SystemPalette LoadFromFile(string path)
    {
        using var stream = new FileStream(path, FileMode.Open);
        using var reader = new BinaryReader(stream);
        var data = reader.ReadBytes(ColorsAmount * 3);
        return new SystemPalette(data);
    }
}
```

Лістинг 6 – код інтерфейсу IMapper.cs

```
namespace Emulator.Mappers;
```

```
public interface IMapper
{
    public void Init(byte[] prg, byte[] chr, int prgMask, int chrMask);
    public byte SelectedBank { get; set; }
    public byte MapperId { get; }
    public byte ReadPrg(ushort addr);
    public byte ReadChrRom(ushort addr);
    public byte ReadChrRam(ushort addr);
    public void WriteChrRam(ushort addr, byte data);
}
```

Лістинг 7 – код абстрактного класу Mapper.cs

```
namespace Emulator.Mappers;

public abstract class Mapper : IMapper
{
    protected byte[] PrgData;
    protected byte[] ChrRomData;
    protected byte[] ChrRamData;

    protected int PrgMask;
    protected int ChrMask;

    public void Init(byte[] prg, byte[] chr, int prgMask, int chrMask)
    {
        PrgData = prg;
        ChrRomData = chr;
        PrgMask = prgMask;
        ChrMask = chrMask;
        ChrRamData = new byte[Rom.CHR_BANK_SIZE];
    }

    public byte SelectedBank { get; set; }

    public abstract byte TotalBanks { get; }
    public abstract byte MapperId { get; }

    public byte ReadChrRom(ushort addr)
```

```
    {
        return ChrRomData[(ushort) (addr & ChrMask)];
    }

    public byte ReadChrRam(ushort addr)
    {
        return ChrRamData[addr];
    }

    public void WriteChrRam(ushort addr, byte data)
    {
        ChrRamData[addr] = data;
    }

    public abstract byte ReadPrg(ushort addr);
}
```

Додаток Б – Диск з роботою