Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська кваліфікаційна робота

на тему: Розробка часового відслідковування дій для спідрану у вигляді моду з використанням Fabric API та MMT

Виконав: студент 4 курсу
групи 3ПІ-21б.
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Туровець Артем Володимирович
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ Катєльніков Д. І.
(прізвище та ініціали)
Рецензент:. к.т.н., доц. каф. ЗІ Дудатьєв А. В.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедри ПЗ
д. т. н. проф. Романюк О. Н.
«03» 06 2025 р.

ВНТУ – 2025

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
О. Н. Романюк
«21» березня 2025 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Туровцю Артему Володимировичу

1. Тема роботи – Розробка часового відслідковування дій для спідрану у вигляді моду з використанням Fabric API та MMT. Керівник: Катєльніков Денис Іванович, викладач кафедри ПЗ, затверджено наказом навчального закладу від 20 березня 2025 р. №97.
2. Термін подання роботи – 2 червня 2025 року.
3. Вихідні дані: специфікації Minecraft (версія 1.20+), середовище IntelliJ IDEA, Fabric API, Minecraft Mapping Tools (MMT), структури даних подій гри, вимоги до точності спідрану, формати обміну даними (CSV/JSON), приклади категорій проходження.
4. Зміст текстової частини: аналіз аналогів; обґрунтування вибору архітектури таймера; постановка задачі; опис структури системи; розробка інтерфейсу та алгоритмів функціонування; реалізація таймера як внутрішньоігрового моду; обробка подій через API; експорт результатів спідрану; тестування функціональності, продуктивності та надійності.
5. Перелік ілюстративної частини: UML-діаграми діяльності та класів; блок-схеми обробки подій; інтерфейс користувача моду; результати тестування у вигляді таблиць та графіків; приклади виводу таймера в інтерфейсі гри

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1-4	Катєльніков Д. І., к.т.н., доц. каф. ПЗ	24.03.25	01.06.25

7. Дата видачі завдання 24 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітк
1	Аналіз існуючих таймерів для ігор та постановка задачі	25.03.25 – 08.04.25	Вик.
2	Розробка архітектури таймера та інтерфейсу користувача	09.04.25 – 30.04.25	Вик.
3	Реалізація обробки подій Minecraft через Fabric API	31.04.25 – 18.05.25	Вик.
4	Реалізація функцій налаштування, запуску, зупинки, збереження	19.05.25 – 22.05.25	Вик.
5	Тестування програмного модуля, оформлення БКР	23.05.25– 30.05.25	Вик.

Студент _____ **А. В. Туровець**
(підпис) (ініціали та прізвище

Керівник бакалаврської кваліфікаційної роботи _____ **Д. І. Катєльніков**
(підпис) (ініціали та прізвище

АНОТАЦІЯ

У роботі розглядається процес розробки модифікації Speedrun Timer для гри Minecraft, призначеної для точного відслідковування часу проходження у режимі спідрану. З метою досягнення високої точності та стабільності було реалізовано внутрішньоігровий таймер, який повністю інтегрується у ігровий цикл завдяки використанню Fabric API, Minecraft Tick System та засобів Minecraft Mapping Tools (MMT). Розроблений модуль забезпечує автоматичний запуск і зупинку таймера, фіксацію подій гри, підтримку категорій спідрану, а також збереження результатів у форматах JSON та CSV. Архітектура програми побудована на модульному принципі з використанням подійної моделі. Проведено тестування системи на точність, продуктивність та сумісність з іншими модами. Запропоноване рішення може бути використане як зручний інструмент для гравців-спідранерів та як основа для подальших розробок у сфері ігрової аналітики.

Ключові слова: Minecraft, спідран, таймер, Fabric API, внутрішньоігрові події, MMT, JSON, модифікація.

ABSTRACT

This thesis presents the development of a Speedrun Timer modification for the game Minecraft, aimed at accurately tracking in-game time during speedruns. To ensure high precision and reliability, an in-game timer was implemented using the Fabric API, Minecraft Tick System, and Minecraft Mapping Tools (MMT). The developed module enables automatic start/stop of the timer, detection of in-game events, support for various speedrun categories, and export of session results in JSON and CSV formats. The architecture is based on a modular and event-driven approach. The system has been tested for accuracy, performance, and compatibility with other mods. The solution serves as a useful tool for speedrunners and provides a foundation for further research and development in the area of game analytics.

Keywords: Minecraft, speedrun, timer, Fabric API, in-game events, MMT, JSON, mod

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. У сучасному світі відеоігри стали невід'ємною частиною культури та повсякденного життя мільйонів людей. Серед них особливу нішу займають ігри з відкритим світом, які надають гравцям необмежені можливості для досліджень, творчості та змагань. Однією з найпопулярніших ігор такого типу є Minecraft – гра, що поєднує простоту візуального оформлення з глибоким ігровим процесом. Вона стала основою для численних ігрових спільнот та зародження унікальних змагальних дисциплін, таких як спідран.

Спідран (скорочено від англ. Speedrunning) передбачає проходження гри або її частини на швидкість із використанням оптимізованих маршрутів, механік гри та стратегій. Участь у подібних змаганнях сприяє розвитку вміння концентрувати увагу на головному, швидкої реакції, кмітливості та спостережливості у гравця. Ці якості є важливими для операторів дронів, операторів телеуправління та робітників багатьох інших сучасних професій.

Гравці з усього світу прагнуть досягати найкращих результатів, для чого використовуються різні інструменти та таймери, що відстежують проміжний та кінцевий час проходження гри Minecraft. Таймери можуть бути частиною гри – так звані інтегровані таймери – або реалізовані незалежними засобами. Інтегровані таймери дають більш точні вимірювання, адже на них впливають ті самі ігрові артефакти, що і на *ігровий світ*. «Ігровий світ» або «Game world» [1] зазвичай визначається як комп'ютерно змодельоване середовище, яке може бути населене багатьма користувачами, що взаємодіють через аватари. Ці світи мають власні правила, фізику та об'єкти, зі своїми правилами взаємодії.

Це робить актуальною розробку інтегрованого таймера для спідрану для гри Minecraft у вигляді спеціально створеної для змагання арени на основі Fabric API, що дозволить зробити підрахунок часу більш точним і нечутливим до зовнішніх факторів. У ігровій індустрії подібні заздалегідь сконфігуровані арени називаються *модами*.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою даної роботи є підвищення точності часових вимірів процесу спідрану у грі Minecraft за рахунок розробки інтегрованого таймера, що працюватиме у вигляді моду з використанням Fabric API. Це дозволить автоматизувати процес фіксації часу проходження, зменшити вплив зовнішніх факторів на процес вимірювання та зробить спідран зручнішим для користувачів.

Для досягнення поставленої мети необхідно виконати такі завдання:

- проаналізувати існуючі рішення для фіксації часу у Minecraft-спідрану;
- визначити оптимальні технології для реалізації таймера;
- розробити архітектуру та логіку роботи системи;
- реалізувати функціонал автоматичного старту, паузи та завершення таймера;
- інтегрувати систему відстеження ігрових подій та тригерів для коректного підрахунку часу;
- організувати взаємодію між елементами системи (GUI, логіка таймера, обробка подій);
- провести тестування та оптимізацію для зменшення навантаження на ресурси гри.

Об’єкт дослідження – процес отримання часових вимірів в процесі спідрану у грі Minecraft.

Предмет дослідження – методи та алгоритми розробки інтегрованого таймера для спідрану для гри Minecraft у вигляді спеціально створеної для змагання арени на основі Fabric API.

Методи дослідження. У роботі використовуються методи об'єктно-орієнтованого програмування та шаблони проектування для розробки архітектури системи; методи оптимізації для розподілу ресурсів для зменшення

навантаження на ігрове середовище; комп’ютерне моделювання для аналізу та перевірки отриманих теоретичних положень.

Новизна отриманих результатів.

1. Подальшого розвитку отримав метод побудови таймеру спідрану, який, на відміну від існуючих методів побудови, створює таймер як частину ігрового світу, що дозволяє синхронізувати його з ігровими процесами і, таким чином, досягати більшої точності і уникати впливу зовнішніх факторів.

2. Подальшого розвитку отримав метод інтеграції таймерів у ігровий світ, який на відміну від існуючих методів, використовує розроблену систему модульної інтеграції, що дозволяє легко адаптувати таймер під різні категорії спідрану.

Практична цінність отриманих результатів. Практична цінність роботи полягає в тому, що на основі отриманих в бакалаврській кваліфікаційній роботі теоретичних положень запропоновано алгоритми та розроблено програмний засіб - таймер для спідранерів Minecraft.

Особистий внесок здобувача. Усі наукові результати, викладені у роботі, отримані особисто автором. У друкованій праці [2] автору належать такі результати: концепція, архітектура та реалізація інтегрованого таймера у вигляді моду (Fabric) для Minecraft, що дозволяє автоматично фіксувати час проходження гри. У друкованій праці [3] автору належать такі результати: реалізація системи збору аналітичної інформації про проходження спідрану та експорт результатів у зовнішні формати.

Апробація матеріалів бакалаврської кваліфікаційної роботи. Описані результати доповідалися на IV Всеукраїнській науково–технічній конференції молодих вчених, аспірантів та студентів «Комп'ютерні ігри і мультимедіа як інноваційний підхід до комунікації – 2024» 26-27 вересня 2024 р. у м. Одеса, а також на Всеукраїнській науково-технічній конференції підрозділів Вінницького національного технічного університету (НТКП ВНТУ–2025).

Публікації. За тематикою дослідження опубліковано у тезах доповідей [2,3].

1 АНАЛІЗ РОЗВИТКУ ТЕХНОЛОГІЙ ЧАСОВОГО ВІДСЛІДКОВУВАННЯ ДІЙ У СПІДРАНАХ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Аналіз стану технологій інформаційних систем для часового відслідковування дій

Спідран як особливий формат проходження відеоігор на швидкість виходить далеко за межі простої розваги. Для багатьох людей участь у спідранах стає важливим інструментом розвитку навичок концентрації, швидкого прийняття рішень та стратегічного мислення. Спідран сприяє покращенню координації рухів, тренує витривалість, увагу до деталей та вміння працювати в умовах обмеженого часу. Крім того, він допомагає розвивати послідовність у досягненні цілей, самодисципліну та вміння аналізувати власні помилки для подальшого вдосконалення.

Розвиток інформаційних технологій кардинально змінив підхід до фіксації часу у спідрані, де кожна секунда може мати вирішальне значення. Якщо на ранніх етапах цього феномену гравці покладались на звичайні механічні або цифрові секундоміри, що вимагали ручної активації, то згодом на зміну цим архаїчним методам прийшли спеціалізовані програмні рішення. Така еволюція відбулася не лише завдяки зростанню вимог до точності, а й через поступове формування професійного середовища, у якому на перший план вийшли об’єктивність результатів, автоматизація процесів та можливість детального аналізу проходження.

Програмні системи, створені спеціально для потреб спідранерів, стали важливим кроком уперед. Вони дозволяють не лише зафіксувати загальний час проходження гри, а й розбити його на окремі сегменти, що дає змогу аналізувати ефективність кожного з них. Наприклад, LiveSplit, одна з найпопулярніших програм, забезпечує користувачів функціоналом для автоматичного таймінгу, синхронізації з відеозаписами та підтримкою численних ігор. Цей інструмент також дозволяє інтегрувати зовнішні API для

обміну даними з платформами на кшталт speedrun.com [4]. Аналогічно, подібні рішення орієнтовані на користувачів, які віддають перевагу легкій у використанні, але водночас потужній системі трекінгу, що не перевантажує ресурси ПК.

Справжнім проривом у галузі стало впровадження внутрішньоігрових таймерів, які оперують не загальним часом гри, а лише активною фазою геймплею, виключаючи екрани завантаження, паузи або технічні затримки. Такий принцип дозволяє досягнути максимальної об’єктивності й точності, особливо у випадках, коли йдеться про визнання світових рекордів або встановлення офіційних результатів. У Minecraft, наприклад, такі рішення реалізовано через модифікації типу speedrunIGT, які використовують дані з внутрішньої пам’яті гри для формування надійних і контрольованих результатів, тим самим уникаючи впливу людського фактору або стороннього ПЗ.

Ще одним важливим напрямом розвитку є інтеграція таймерів із системами збору статистики. Деякі сучасні програмні модулі здатні не лише відслідковувати час, а й проводити комплексний аналіз проходження, включаючи побудову графіків, визначення вузьких місць у маршруті та аналіз частоти використання технік. Все це дозволяє не просто спостерігати за результатами, а системно вдосконалювати маршрут, методику та навіть ігрові звички. Рішення, які підтримують збереження даних у форматах CSV, JSON або навіть пряме підключення до баз даних, дають гравцям інструменти аналітики, які раніше були доступні лише в професійному кіберспорті.

Окрему роль відіграють глобальні онлайн-платформи, такі як Speedrun.com, які не тільки зберігають результати, а й виконують верифікацію проходжень за допомогою відеодоказів, автоматизованих перевірок і детальних регламентів. Це створює справді конкурентне середовище, де результати повинні бути не лише швидкими, а й беззаперечно підтвердженими. Такий підхід формує нову етику у сфері цифрового змагання – чесність, прозорість і точність стають ключовими стандартами.

Разом із цим, навіть у технологічно просунутому середовищі залишаються нерозв’язані питання. Серед основних викликів – сумісність таймерів із різними версіями ігор, наявність внутрішньоігрових відмінностей, що впливають на механізми запуску/завершення сегментів, а також варіативність платформи: Windows, Linux, macOS. Постійна потреба адаптувати програмне забезпечення до нових умов змушує розробників працювати у режимі постійного оновлення. Крім технічних аспектів, важливою проблемою залишається і верифікація результатів. Різні версії одного й того ж таймера можуть інтерпретувати одні й ті ж події по-різному, що призводить до суперечок у спільноті.

Особливу надію на вирішення цих проблем покладають на новітні технології, зокрема штучний інтелект. Уже сьогодні існують експериментальні модулі, які з допомогою нейронних мереж автоматично аналізують відеозаписи спідранів, визначають порушення правил, оцінюють ефективність руху гравця по маршруту. У перспективі це дозволить значно прискорити процес перевірки проходжень та вивести систему підтвердження результатів на якісно новий рівень.

Зважаючи на описані тенденції, створення власного таймера Speedrun Timer у межах цього проєкту стало логічним кроком до удосконалення внутрішньоігрової системи відстеження часу.

Розроблюваний модуль, буде враховувати внутрішню логіку гри, і при відповідній кастомізації інтерфейсу та категорій, буде представляти собою продукт, який стане ефективним інструментом у розпорядженні як новачків, так і досвідчених спідранерів. Така система дозволить не лише фіксувати результати, але й глибоко аналізувати проходження, виводячи процес на рівень повноцінного цифрового дослідження.

Таким чином, інструменти фіксації часу у спідранінгу зазнали значної еволюції, перетворившись із простих секундомірів на високотехнологічні аналітичні системи. Очевидно, що розвиток триватиме й надалі, охоплюючи все

більше аспектів проходження гри, формуючи нові стандарти точності, зручності й надійності.

1.2 Аналіз та порівняння аналогів

У сучасних умовах цифрової епохи спідранінг перестає бути просто захопленням і перетворюється на повноцінний формат змагань, який вимагає від учасників не лише ігрової майстерності, а й технічної точності. Відповідно до цього зростає потреба у високоточних програмних рішеннях для фіксації часу проходження, які б відповідали сучасним вимогам спільноти. Розробка власного таймера потребує ретельного вивчення вже наявних програмних продуктів, що використовуються у сфері спідранів. Саме тому першим і необхідним етапом є комплексний аналіз існуючих рішень, який дозволяє не лише зрозуміти їхню архітектуру та функціонал, але й виявити прогалини, що можуть бути вдосконалені або усунені в рамках власної розробки.

Порівняння з аналогами дозволяє сформувати чітке бачення потреб користувачів і тенденцій, які сьогодні переважають на ринку інструментів для відстеження часу в іграх. Основними критеріями, на які варто звернути увагу під час такого аналізу, є точність фіксації часових сегментів – окремих проміжків проходження гри, а також здатність системи реагувати на внутрішньоігрові події, наявність підтримки автоматичних тригерів, легкість у використанні під час гри, а також можливість глибокої кастомізації інтерфейсу та інтеграції з онлайн-платформами, такими як speedrun.com.

Окрему увагу заслуговує питання сумісності таких інструментів із різними операційними системами та ігровими платформами. Рішення, які надають універсальність у використанні, автоматично стають більш привабливими для ширшої аудиторії, особливо якщо йдеться про багатоплатформенне середовище.

Крім функціональних переваг, критичну роль відіграє і продуктивність програмного модуля. Під час швидкісного проходження гри будь-які затримки або збої у роботі таймера можуть не лише вплинути на результат, але й

зіпсувати весь процес гри. Саме тому важливо забезпечити стабільність відображення таймеру, плавність оновлення інформації на екрані, а також мінімізацію споживання системних ресурсів.

Для забезпечення повноти оцінки, у процесі аналізу було розглянуто кілька найпопулярніших рішень, серед яких LiveSplit, SpeedrunIGT, Splitty та WSplit. Кожен з цих інструментів має свої характерні риси, що сформували навколо них активну спільноту користувачів. У ході аналізу було виявлено не лише їхні переваги, зокрема простоту у використанні або можливості автоматичної синхронізації з платформами, а й недоліки – відсутність підтримки модифікацій Minecraft або надмірно складний інтерфейс налаштувань.

Таким чином, проведене дослідження дає змогу зробити висновок про доцільність розробки нового модуля, що буде поєднувати сильні сторони вже існуючих рішень і водночас усуватиме типові недоліки. Це дозволить створити дійсно функціональний, інтуїтивно зрозумілий та продуктивний таймер, орієнтований на потреби сучасного спідран-ком’юніті Minecraft.

LiveSplit [5] – один із найпопулярніших таймерів серед спідранерів, що підтримує широкий набір функцій для точного відстеження часу під час проходження ігор. Він дозволяє налаштовувати вигляд і розташування елементів таймера на екрані відповідно до потреб користувача. Однією з ключових функцій є автоматичні тригери – спеціальні механізми, які самостійно фіксують важливі події у грі, наприклад, початок рівня або перемогу над босом, що знімає необхідність ручного перемикання етапів.

Важливою особливістю також є підтримка сплітів – розподілу забігу на сегменти, кожен з яких відповідає певному досягненню в грі. Це допомагає гравцям аналізувати втрати часу на окремих етапах і вдосконалювати свої результати. Однак, налаштування програми потребує часу та досвіду, а також вона працює лише на платформі Windows (див. рисунок 1.1).

Рисунок 1.1 – Таймер LiveSplit

Основні переваги та недоліки спідранерського таймера Live Split описано у таблиці 1.1.

Таблиця 1.1 – Переваги та недоліки таймера LiveSplit

Переваги	**Недоліки**
Гнучкі налаштування інтерфейсу	Вимагає ручного налаштування
Підтримка автоматичних тригерів	Обмежена підтримка платформ (Windows)
Інтеграція з Twitch та іншими сервісами	Відносно складний для новачків

WSplit [6] – простий та легкий таймер, розроблений для швидкої роботи без зайвих функцій. Він має мінімалістичний інтерфейс та низьке споживання

ресурсів, що робить його хорошим варіантом для старих або слабких комп'ютерів. Проте, цей таймер не підтримує автоматичні тригери та має обмежені можливості кастомізації (див. рисунок 1.2).

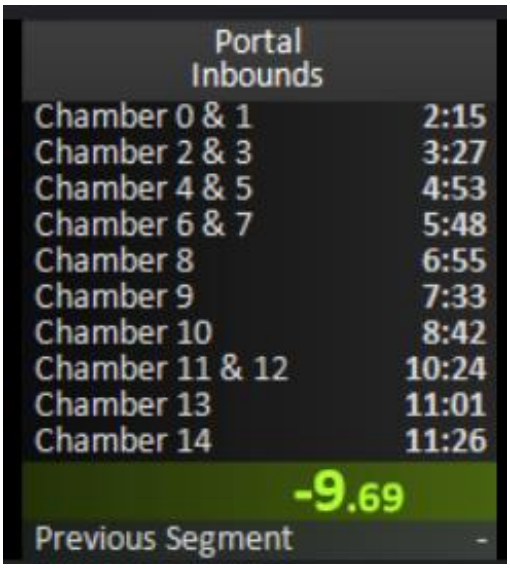


Рисунок 1.2 – Таймер WSplit

Основні переваги та недоліки спідранерського таймера WSplit описано у таблиці 1.2.

Таблиця 1.2 – Переваги та недоліки таймера WSplit

Переваги	**Недоліки**
Простота використання	Відсутність підтримки автоматичних тригерів
Легкість та низьке споживання ресурсів	Обмежені можливості кастомізації
Мінімалістичний інтерфейс	Відсутність інтеграції з іншими сервісами

Splitty [7] – цекросплатформенний таймер, який працює на різних операційних системах, включаючи Windows, macOS та Linux. Він підтримує

налаштування гарячих клавіш для зручного використання, проте має обмежені можливості для кастомізації та не підтримує тригери (див. рисунок 1.3).

Рисунок 1.3 – Таймер Splitty

Основні переваги та недоліки спідранерського таймера Splitty описано у таблиці 1.3.

Таблиця 1.3 – Переваги та недоліки таймера Splitty

Переваги	**Недоліки**
Підтримка різних платформ	Відсутність широких можливостей кастомізації
Можливість налаштування гарячих клавіш	Відсутність підтримки автоматичних тригерів
Простий у налаштуванні	Обмежена інтеграція з іншими сервісами

SpeedrunIGT [8] – це таймер, орієнтований на точне вимірювання внутрішньоігрового часу (IGT). Він використовує автоматичні тригери для фіксації точного часу спідрана. Однак, цей таймер має вузьку спеціалізацію, що обмежує його використання в різних іграх (див. рисунок 1.4).

Рисунок 1.4 – Таймер SpeedrunIGT

Основні переваги та недоліки спідранерського таймера SpeedrunIGT описано у таблиці 1.4.

Таблиця 1.4 – Переваги та недоліки таймера SpeedrunIGT

Переваги	**Недоліки**
Висока точність відстеження часу	Вузька спеціалізація
Підтримка автоматичних тригерів	Обмежена підтримка ігор
Мінімальне втручання користувача	Відсутність кастомізації

Таким чином, аналіз існуючих таймерів дозволяє виявити їхні сильні та слабкі сторони, що важливо для розробки нового рішення, яке усуває недоліки аналогів та забезпечує широкий функціонал для користувачів.

Для кращого розуміння переваг розробленого рішення його було порівняно з аналогами за ключовими критеріями (див. таблицю 1.5).

Таблиця 1.5 – Порівняльна характеристика аналогів та власної розробки

Критерій	Live Split	WSplit	Splitty	Speedrun IGT	Власний продукт
Точність часу	Висока	Середня	Середня	Висока	Висока
Автоматичні тригери	Частково	Ні	Ні	Так	Так, розширені можливості
Кросплатформенність	Ні	Ні	Так	Ні	Так
Кастомізація	Висока	Низька	Середня	Середня	Висока
Інтеграція з онлайн-сервісами	Так	Ні	Ні	Так	Так, з розширеним функціоналом

На основі проведеного аналізу можна зробити висновок, що розроблений таймер Speedrun Timer поєднує в собі найкращі функції існуючих аналогів, усуваючи їхні основні недоліки. Він має високу точність відстеження часу, підтримує автоматичні тригери та сумісний із різними платформами. Крім того, передбачена широка кастомізація інтерфейсу та гнучкі можливості інтеграції з онлайн-сервісами, що робить його привабливим для широкого кола спідранерів. Завдяки всім цим особливостям Speedrun Timer є оптимальним

вибором для користувачів, які прагнуть максимально точно та зручно фіксувати свої результати.

1.3 Дослідження методів вирішення задачі

Розробка кастомізованого таймера для спідранів у Minecraft вимагає комплексного підходу, що включає створення гнучкого інтерфейсу, налаштування категорій спідранів, точного відображення часу сегментів, збору та аналізу статистики, а також оптимізації продуктивності. Для реалізації цих завдань необхідно дослідити сучасні методи розробки програмного забезпечення для ігор, обробки реального часу та роботи з графічним інтерфейсом.

Головною вимогою до інтерфейсу є його інтеграція в гру без значного навантаження на систему. Основними підходами є використання Fabric API та OpenGL для створення графічних елементів. Інтерфейс може бути реалізований за допомогою UI-компонентів, збережених у JSON-конфігураціях, що дозволяє користувачам змінювати зовнішній вигляд таймера. Також варто розглянути можливість використання Canvas Renderer – інструменту, який дозволяє малювати кастомні графічні елементи безпосередньо у грі. Він працює як шар поверх зображення гри, на якому можна програмно створювати та відображати текст, лічильники, іконки чи інші графічні об'єкти без зміни основного ігрового контенту. Це відкриває широкі можливості для створення зручного інтерфейсу таймера або інших візуальних підказок для спідранерів. [9].

Різні категорії спідранів вимагають різних умов проходження, що повинно враховуватися у налаштуваннях таймера. Оптимальним методом є використання конфігураційних файлів у форматах JSON або YAML, які дозволяють визначати параметри кожної категорії, наприклад, список дозволених дій або використання певних стратегій. Альтернативою є інтеграція бази даних, що дозволяє користувачам ділитися налаштуваннями через сервер [10].

Для точного підрахунку часу проходження важливо відстежувати події в грі. Один із підходів – використання внутрішньоігрових тригерів, які можуть викликати відповідні функції при проходженні певних точок маршруту.

У цьому контексті важливу роль відіграє використання MMT (Minecraft Mapping Tools) – набору інструментів і методик, що дозволяють програмно звертатися до структур гри, відстежувати координати, блоки, сутності та події у внутрішньому ігровому середовищі Minecraft [11]. Завдяки інтеграції MMT розроблений таймер отримує доступ до детального опису ігрового стану, що дає змогу точно визначати моменти старту, завершення сегментів або переходів між фазами проходження. Це забезпечує синхронізацію логіки таймера з реальним перебігом подій у грі та мінімізує ризик помилкових або несвоєчасних спрацьовувань.

Алгоритм повинен підтримувати синхронізацію з системним часом та мати низьку затримку. Використання багатопоточності дозволить обробляти події без шкоди для продуктивності гри [12].

Аналіз результатів спідранів є важливою частиною процесу. Для цього необхідно розробити систему логування та збереження результатів проходження. Найефективнішим підходом є використання локальних баз даних, таких як SQLite, або збереження даних у форматах CSV та JSON. Також можна інтегрувати відправку статистики на сервери з підтримкою API, що дозволить порівнювати результати з іншими гравцями [13].

Щоб забезпечити стабільну роботу таймера, необхідно провести комплексне тестування. Для цього використовуються профайлери, такі як Spark або VisualVM, що дозволяють оцінити навантаження на процесор та пам'ять. Основні методи оптимізації включають кешування даних, зменшення кількості запитів до API та ефективне використання багатопоточності для обробки подій у реальному часі [14].

Аналіз методів розробки спідранерських таймерів дозволяє виділити найбільш ефективні підходи до вирішення поставлених завдань. Використання Fabric API та OpenGL для інтерфейсу, збереження налаштувань у JSON/YAML,

багатопотокової обробки подій та баз даних для аналізу статистики забезпечить надійну та продуктивну роботу таймера.

1.4 Постановка задач бакаларської роботи

У бакаларськ роботі передбачається розробка програмного модуля для відображення та аналізу спідранерських результатів у грі Minecraft. Основним завданням дослідження є створення кастомізованого таймера, який буде інтегровано у гру з метою забезпечення точного підрахунку часу проходження. Для досягнення цього необхідно сформулювати низку підзадач, реалізація яких сприятиме побудові повноцінного і функціонального програмного засобу. Це дослідження охоплює як технічні, так і логічні аспекти реалізації, включаючи проектування інтерфейсу, оптимізацію алгоритмів обліку часу, впровадження засобів збору статистики, а також забезпечення стабільності та сумісності програмного модуля з іншими компонентами гри.

Передбачається створення кастомізованого інтерфейсу таймера для спідранів у Minecraft, який має бути органічно інтегрований у візуальне середовище гри, не відволікати гравця від ігрового процесу, водночас забезпечуючи зручність зчитування даних. Візуальна частина реалізується з використанням Fabric API та OpenGL, а конфігураційна гнучкість забезпечується через налаштування у форматах JSON або YAML. Інтерфейс має враховувати як базові потреби новачків, так і вимоги досвідчених користувачів, допускаючи зміну кольорової схеми, шрифтів, анімацій та режимів відображення.

Наступним кроком є створення системи налаштування категорій спідранів. Ця система повинна дозволити користувачеві обирати або задавати параметри категорій безпосередньо через інтерфейс або через редагування конфігураційних файлів. Також передбачається функція обміну налаштуваннями між гравцями, що відкриває нові можливості для персоналізації ігрового досвіду. Така система повинна бути гнучкою та

масштабованою, щоб надалі можна було розширити її можливості без радикального втручання в кодову базу.

Важливим завданням є розробка алгоритму фіксації часу сегментів – процесу точного запису моменту завершення певного етапу гри для подальшого аналізу прогресу. Алгоритм має спиратися на події в ігровому середовищі, що визначають ці моменти, і забезпечувати автоматичну реєстрацію часу без участі гравця, мінімізуючи похибки та затримки у фіксації результатів.

У реалізації цієї функціональності ключову роль відіграє використання Minecraft Mapping Tools, які дозволяють формалізувати внутрішньоігрові умови, що запускають чи завершують сегменти проходження, на основі конкретних змін у світі гри – наприклад, досягнення певних координат, взаємодії з об’єктами або зміни станів гравця.

Застосування багатопоточності дає змогу підвищити точність таймера та знизити затримки при обробці ігрових подій. Крім того, алгоритми повинні враховувати особливості внутрішньоігрової фізики, часові виключення (наприклад, паузи чи завантаження) та інші умови, що впливають на загальну тривалість проходження.

Окрему увагу планується приділити створенню системи збору статистичних даних. Усі результати проходжень мають бути збережені у зручному структурованому вигляді, з можливістю подальшого перегляду, порівняння, аналізу. Дані мають експортуватися у формати CSV або JSON для зручності обробки поза межами гри. Статистичні блоки також повинні враховувати розбивку за спробами, середні показники, кількість невдалих запусків, а також окремо зберігати інформацію про особисті рекорди та тенденції у зміні результатів з часом.

Також буде здійснено поетапне тестування кожного з модулів. Передбачено вивчення впливу модифікації на ресурси комп’ютера, її сумісності з іншими модами, стабільності роботи в реальних умовах ігрового середовища.

Таким чином, постановка задач дослідження включає реалізацію функціонального, продуктивного та зручного у використанні

внутрішньоігрового таймера, який відповідатиме сучасним вимогам спільноти Minecraft-спідранерів і надасть користувачам інструмент для підвищення результативності в межах заданої дисципліни. Очікується, що впровадження цього рішення дозволить підвищити якість та прозорість проходжень, сприятиме популяризації формату спідранів у середовищі гравців та розширить можливості індивідуального аналізу ігрових досягнень.

1.5 Висновки

У ході виконання першого розділу бакаларської роботи було проведено детальний аналіз предметної області, сформульовано мету та завдання дослідження, а також визначено методи їх вирішення. Робота зосереджувалася на розробці кастомізованого таймера для спідранів у Minecraft, що включає такі основні аспекти: інтеграція в ігрове середовище, реалізація системи категорій спідранів, точне відображення часу сегментів, збір і аналіз статистичних даних, а також оптимізація продуктивності.

Аналіз існуючих підходів до створення таймерів для спідранів дозволив виокремити ключові методи та інструменти, які можуть бути використані в розробці. Було обґрунтовано вибір технологій, таких як Fabric API для інтеграції мода в Minecraft, JSON/YAML для збереження конфігурацій, а також бази даних для збору та аналізу статистичних даних.

2 РОЗРОБКА МЕТОДУ ПОБУДОВИ ТАЙМЕРУ СПІДРАНУ

2.1 Аналіз вхідних даних системи

Процес розробки програмного забезпечення для спідранерського таймера в Minecraft є комплексною задачею, яка вимагає не лише знань програмування, але й глибокого аналізу вхідних даних системи. Вхідні дані виступають базовим елементом у функціонуванні таймера, адже саме на їх основі здійснюється обчислення часу проходження, ведення розширеної статистики, формування користувацьких налаштувань і налаштування інтеграції з іншими модулями гри.

Одним із головних джерел вхідних даних є налаштування користувача. Ці конфігураційні параметри визначають поведінку таймера, включаючи формат виведення даних, обробку проміжних результатів і спосіб обліку часу для різних категорій спідранів – Any%, 100% або специфічні модифікації. Налаштування часто зберігаються у форматах JSON або YAML, що забезпечує гнучкість у зміні параметрів, зручність інтеграції з іншими модулями, можливість версіювання налаштувань і легкість автоматичної синхронізації між різними пристроями.

У процесі функціонування таймера ключову роль відіграє збереження результатів у зручних і універсальних форматах. Для цього реалізовано підтримку двох основних типів: JSON та CSV. Формат JSON використовується для структурованого зберігання повної інформації про ігрову сесію, включаючи час кожного сегмента, тип категорії, дані про спроби, параметри користувача та події, які вплинули на хід проходження. Така гнучка структура дозволяє легко відновити повну картину сесії або здійснити глибоку аналітику за допомогою зовнішніх сервісів.

CSV, у свою чергу, застосовується для експорту лінійних таблиць з ключовими числовими значеннями: часу проходження, різниці між спробами, кількості перезапусків. Завдяки простоті та сумісності, цей формат легко імпортується у сторонні аналітичні засоби, такі як Microsoft Excel або Google

Sheets, що є важливою вимогою спільноти спідранерів при звітуванні результатів.

Ще одним важливим аспектом є обробка інформації про події, що відбуваються безпосередньо в ігровому світі. Серед таких подій можна виокремити активацію структур, зміну фаз гри або отримання досягнень, що визначають прогрес гравця.

Для підвищення точності і повноти обробки внутрішньоігрових подій у модулі використано Minecraft Mapping Tools (MMT), що забезпечує програмний доступ до структури світу Minecraft, координат, типів об'єктів і станів. Інтеграція з MMT дозволяє модулю синхронізувати часові мітки з конкретними подіями у світі гри, що значно підвищує точність фіксації моментів запуску, сплітів і завершення спідрану.

Усі ці події фіксуються через внутрішні журнали Minecraft, які надалі обробляються через API Fabric. Це дозволяє не лише інтегрувати таймер безпосередньо у реальний ігровий цикл, а й забезпечити високоточний моніторинг прогресу гравця з мінімальними затримками.

Окрему категорію вхідних даних становлять часові мітки проходження. Вони містять детальну інформацію про час досягнення кожного важливого сегмента гри, що дає змогу будувати гнучкі графіки прогресу, проводити ретельний аналіз спроб, порівнювати результати між різними спробами, а також формувати прогностичні структури поведінки гравця. Такі мітки можуть зберігатися як у простих текстових файлах CSV, так і у більш складних бінарних форматах, що оптимізовані для швидкого зчитування великих обсягів інформації.

Не менш важливою складовою є збір даних про поточний стан гри. Ці дані охоплюють такі показники, як координати та розташування гравця, статуси світу, наявність або використання ресурсів, виконання поточних завдань та подолання ігрових етапів. Аналіз поточного стану дозволяє оперативно

оцінювати коректність проходження спідрану, виявляти потенційні аномалії у грі, а також визначати фактори впливу на кінцевий результат [15].

Для забезпечення точності та узгодженості вхідних даних застосовуються перевірені й вдосконалені методи обробки. По-перше, використовуються високоефективні механізми обробки подій у реальному часі, що дозволяють системі миттєво реагувати на зміни в ігровому середовищі та оперативно оновлювати дані на екрані користувача. По-друге, усі ключові події автоматично логуються у спеціалізовані файли для подальшого глибинного аналізу та аудиту процесу гри. По-третє, збереження конфігураційних файлів і даних про спроби здійснюється у форматах JSON та CSV, що дає змогу швидко отримувати доступ до потрібної інформації через стандартні інструменти обробки даних та сумісність із зовнішніми системами аналітики. JSON використовується для зберігання повної структури сесії, тоді як CSV забезпечує швидкий доступ до ключових числових характеристик.

По-четверте, у системі застосовуються передові алгоритмічні методи фільтрації та оптимізації даних – структурованих процедур, які дозволяють автоматично визначити важливість подій, видалити зайвий шум у даних, зменшити затримки при обробці. Ці процедури спрямовані на мінімізацію затримок у роботі програми, підвищення загальної продуктивності модуля та забезпечення стабільності його роботи навіть за умов високих навантажень [16]. Використання таких методик дозволяє значно покращити досвід користувача та зменшити час реакції таймера.

Аналіз вхідних даних є критично важливою та невід’ємною частиною архітектури системи. Саме від якості та точності цих даних безпосередньо залежить правильність обчислення часу проходження, достовірність статистики та ефективність інтеграції модуля у широке спідран-середовище Minecraft. Завдяки ретельно продуманій структурі збору, збереження, обробки й оптимізації інформації, Speedrun Timer демонструє здатність працювати стабільно, точно й ефективно, забезпечуючи користувачам якісний інструмент

для глибокого аналізу їхніх ігрових сесій та подальшого вдосконалення навичок.

2.2 Розробка інтерфейсу програмного додатку

Під час розробки інтерфейсу програмного застосунку Speedrun Timer головним завданням було забезпечити зручність користування, інтуїтивність навігації та мінімізацію навантаження на користувача під час активного проходження ігрового процесу. Оскільки модуль призначено для ведення часу у спідранах гри Minecraft, інтерфейс мав бути максимально простим, легким для сприйняття та водночас функціональним.

У процесі проектування було вирішено дотримуватися стилістики гри Minecraft, обравши кольорову гаму, характерну для піксельної графіки. Основним рішенням стала темна тема інтерфейсу, яка дозволяє знизити навантаження на очі гравців під час тривалих сесій. Для відображення ключових елементів таймера використовуються яскраві акцентні кольори, такі як зелений, жовтий та червоний, що дозволяють швидко орієнтуватися в основній інформації, не відволікаючись від ігрового процесу.

Для доступу до головного меню моду передбачено простий механізм: користувач має відкрити параметри гри та у правому верхньому куті натиснути на піктограму у вигляді книжки. Це рішення забезпечує логічне вписування моду у загальну навігаційну систему Minecraft та мінімізує необхідність додаткових інструкцій.

При завантаженні Speedrun Timer користувача зустрічає екран із відображенням логотипу програми та її назви «Speedrun Timer», що дозволяє швидко впізнати активний мод (див. рисунок 2.1).

Рисунок 2.1 – Відображення моду у списку

Головне меню містить основні параметри таймера: запуск, пауза, зупинка, налаштування категорій спідрану. Воно розміщене у верхній частині екрану, щоб не заважати ігровому процесу. За допомогою головного меню здійснюється навігація між основними елементами (див. рисунок 2.2).

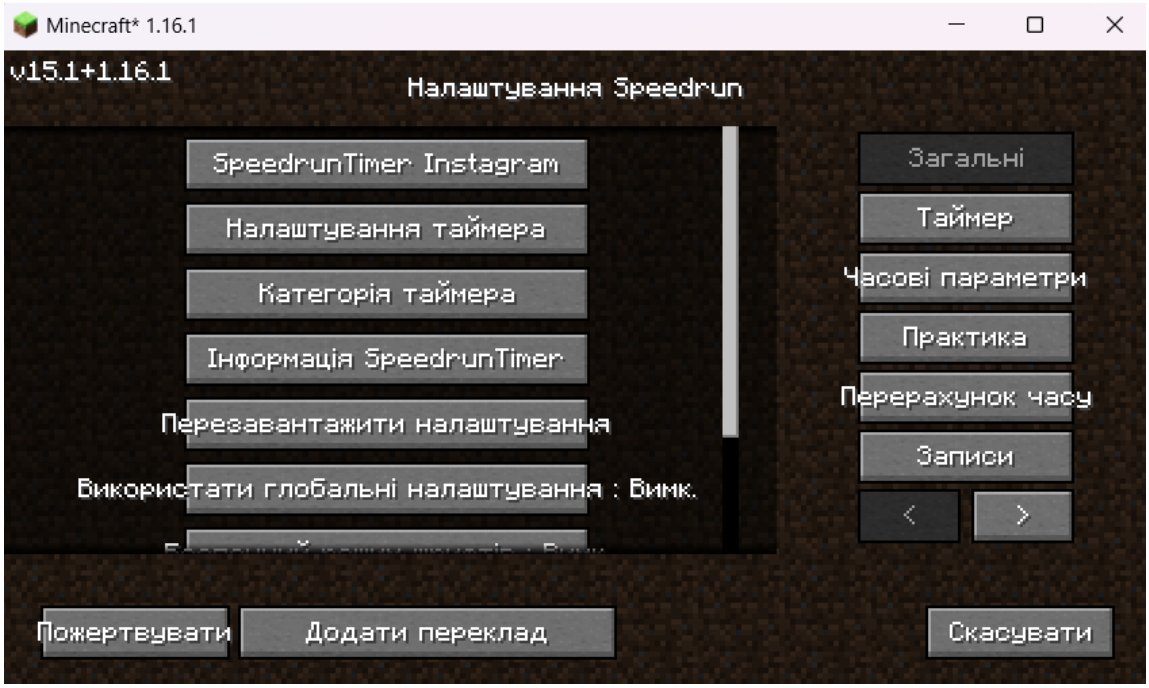


Рисунок 2.2 – Головне меню

Один з головних розділів таймера має назву «Категорія таймера». У цьому розділі користувач може змінювати параметри для проходження спідрану (див. рисунок 2.3).

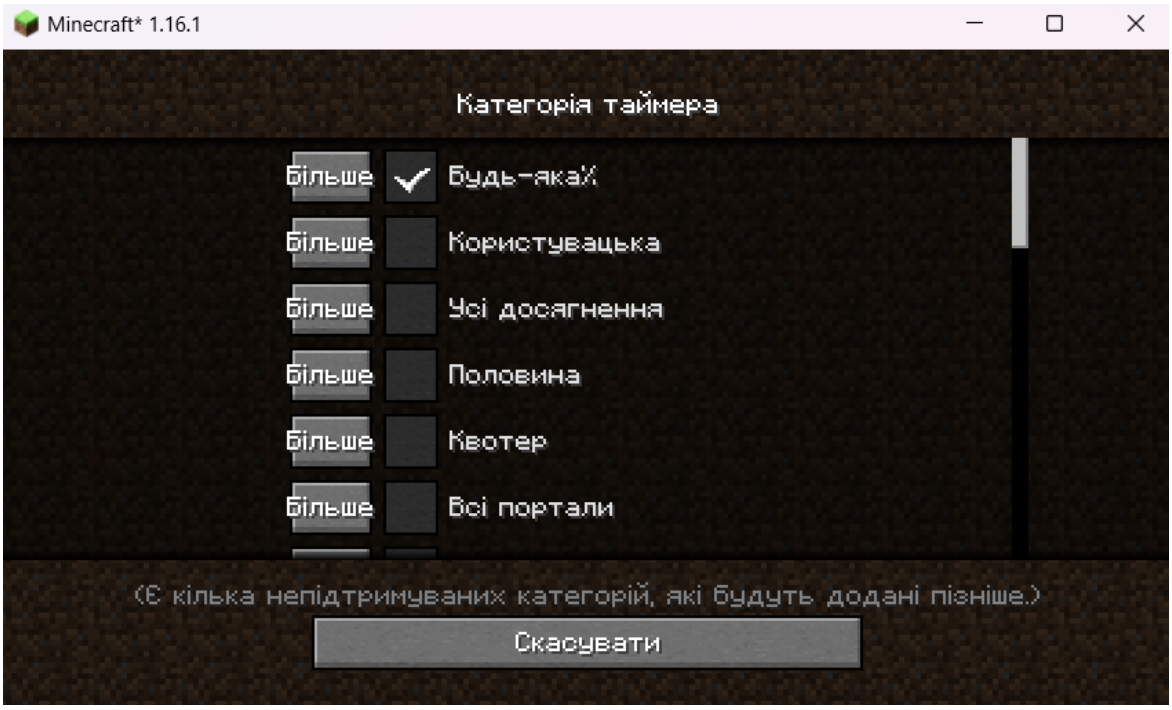


Рисунок 2.3 – Розділ «Категорія таймера»

Функції розділу:

- вибір категорії (наприклад, Any%, Glitchless, Random Seed);
- встановлення гарячих клавіш для старту/паузи/зупинки;
- включення або вимкнення додаткової статистики.

Для того, щоб змінити налаштування таймера, потрібно натиснути на кнопку «Налаштування таймера», після чого користувача перекине на сторінку кастомізації (див. рисунок 2.4).

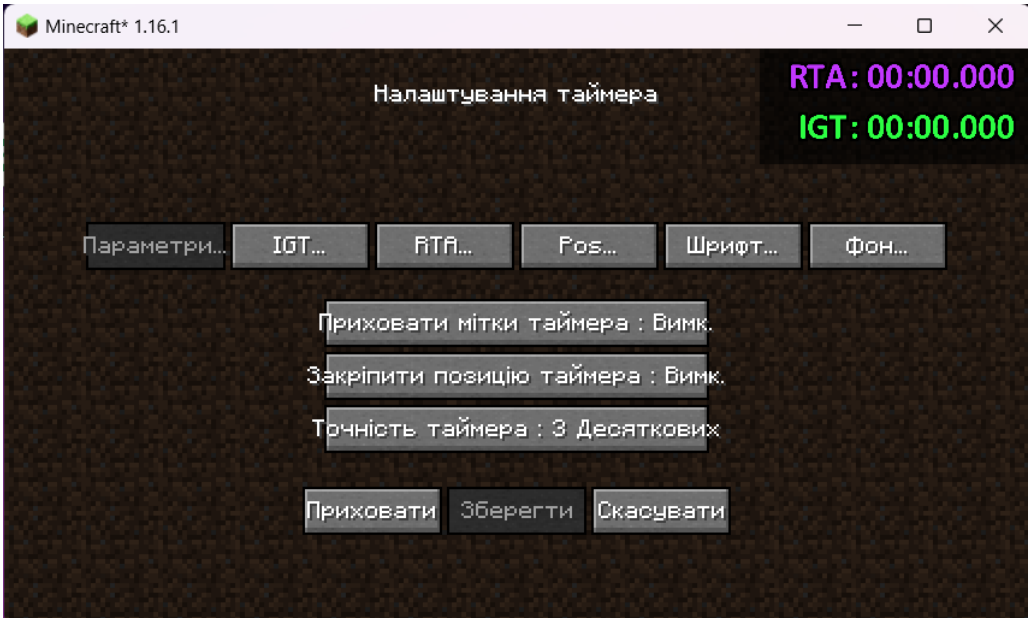


Рисунок 2.4 – Розділ налаштування таймера

У налаштуваннях користувач може змінити колір таймера, шрифт тексту, розташування основного блоку, а також стиль відображення. Це все забезпечується конкретними налаштуваннями кожного конкретного елемента та надає можливість створити свій унікальний варіант (див. рисунок 2.5).

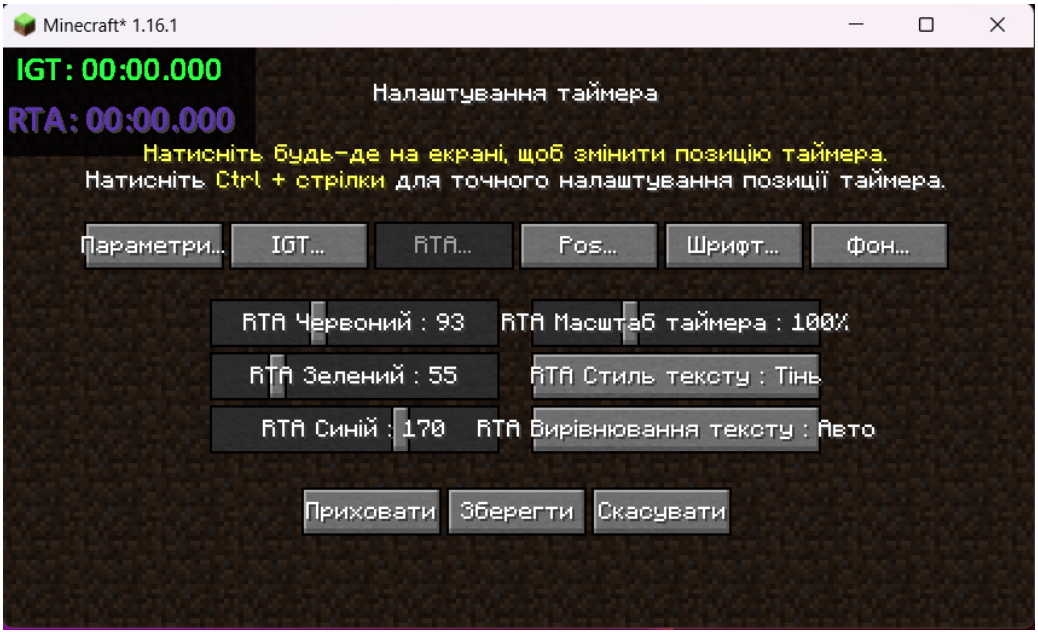


Рисунок 2.5 – Кастомізація таймера

Після завершення спідрану користувач може переглянути розширену статистику проходження. Вікно включає:

1. Графічне відображення часу на кожному етапі.
2. Порівняння з попередніми спробами.
3. Функцію експорту результатів у форматі JSON (див. рисунок 2.6).

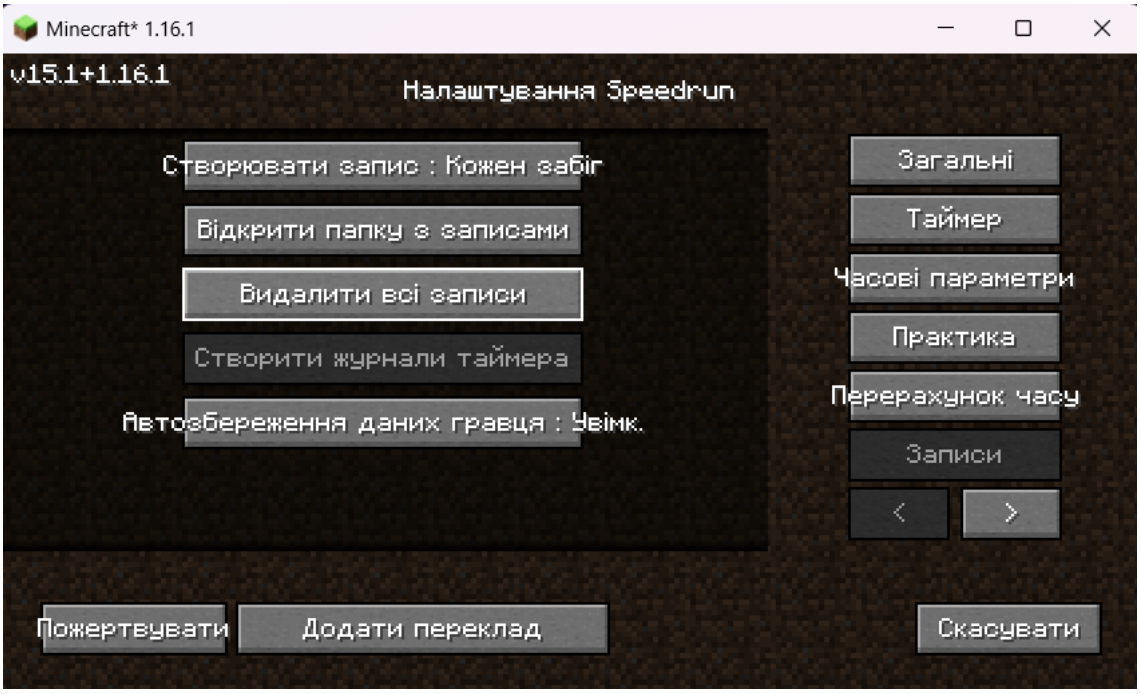


Рисунок 2.6 – Розділ записів

Також міститься окремий відділ, який містить опис усіх функцій Speedrun Timer, інструкцію з використання, а також інформацію про версію моду та розробника.

Інтерфейс було розроблено у середовищі розробки IntelliJ IDEA з використанням технологій Fabric API, Java та бібліотек для рендерингу графічних елементів Minecraft. Візуальні компоненти створювалися з урахуванням вимог до продуктивності, щоб не впливати на ігровий процес.

2.3 Розробка структури та алгоритмів роботи системи

Одним з найважливіших кроків є перевірка наявності необхідних компонентів мода та його коректного завантаження. Якщо мод не було встановлено або знайдено критичні помилки, програма виведе повідомлення про помилку. У разі успішного завантаження користувач переходить до головного меню, де може обрати налаштування відображення таймера, зміну його позиції, форматування тексту, категорії спідрану.

Після запуску гри таймер автоматично починає відлік часу відповідно до вибраної категорії. Якщо активовано функцію очікування першого введення (First Input), таймер розпочнеться тільки після першої дії гравця. У процесі гри таймер постійно відображає поточний час проходження, включаючи проміжні відрізки часу та зміни, що відбуваються під час проходження спідрану.

Користувач може зупинити, перезапустити або змінити формат відображення таймера безпосередньо в меню модифікації. Якщо вибрана категорія спідрану передбачає конкретні події, мод автоматично визначає момент їхнього настання та фіксує їх у статистиці проходження. Це стало можливим завдяки застосуванню Minecraft Mapping Tools, які забезпечують глибокий доступ до стану гри та обробку логіки подій із високою точністю.

Для кращого розуміння роботи модулів програмного засобу Speedrun Timer було вирішено розробити структуру роботи системи на прикладі UML-діаграми діяльності. Згідно з вказаною діаграмою користувач для початку взаємодії із програмним застосунком повинен запустити Minecraft із встановленим модом Speedrun Timer (див. рисунок 2.7).

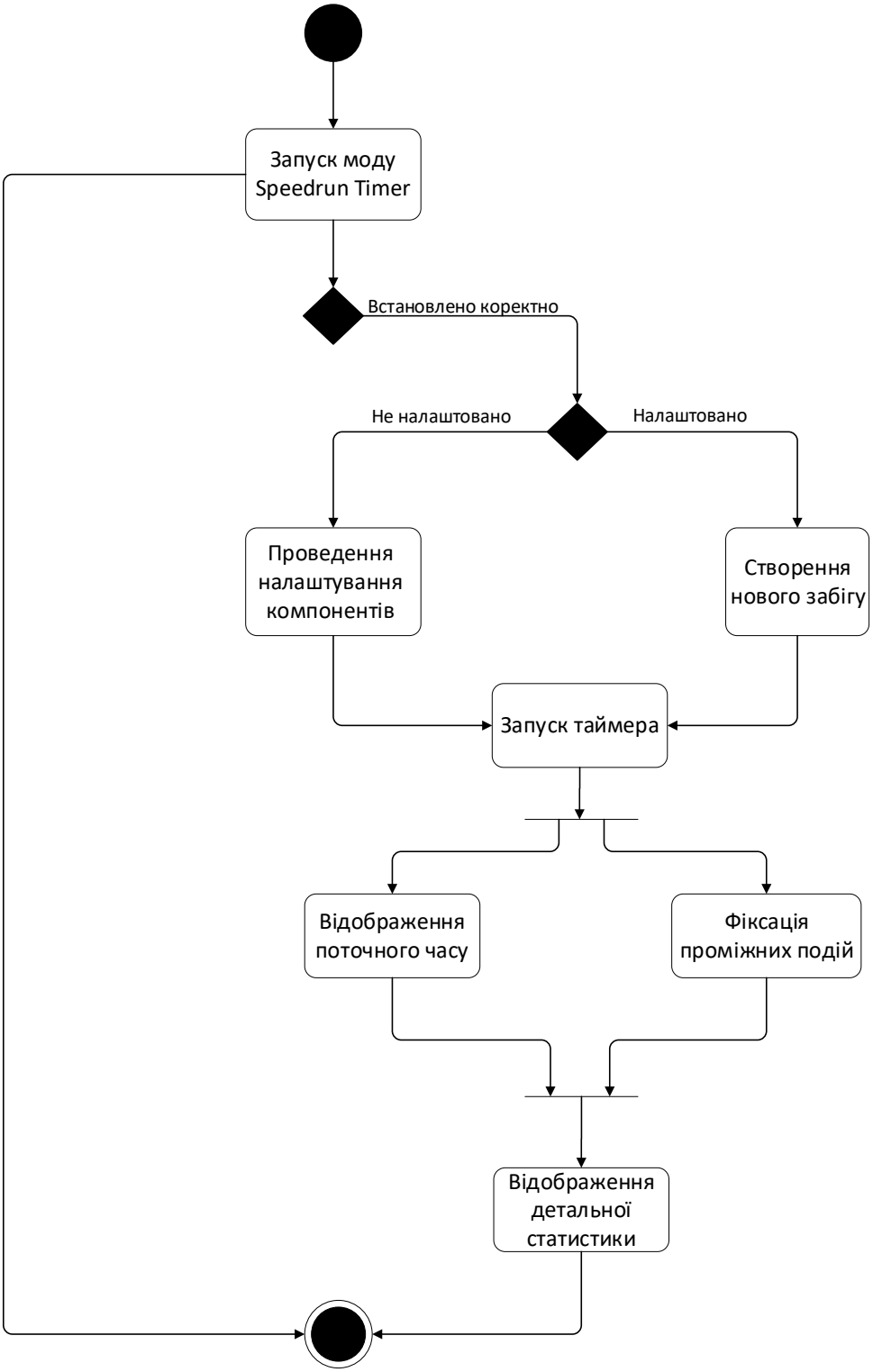


Рисунок 2.7 – Діаграма діяльності

Після завершення спідрану користувач отримує детальний аналіз проходження: загальний час, проміжні відтинки, кількість перезапусків, графічне відображення часу проходження. Також доступна можливість

експорту отриманих результатів у форматі .csv або .json для подальшого аналізу та подачі на speedrun.com.

В результаті створено діаграму діяльності програмного засобу, яка детально описує загальний алгоритм роботи модифікації. Розробка UML-діаграми діяльності була виконана у програмному забезпеченні Visio [17].

Не менш важливим етапом є створення алгоритмів роботи продукту. На першому етапі користувач завантажує Minecraft із встановленим модом Speedrun Timer. Після запуску гри здійснюється перевірка коректності завантаження мода, наявності всіх необхідних компонентів та їхньої сумісності з поточною версією гри. У разі виявлення критичних помилок або відсутності компонентів програма виводить відповідне повідомлення про помилку. Якщо всі компоненти коректні, система переходить до головного меню.

У головному меню користувач може налаштувати параметри таймера: вибрати категорію спідрану, встановити позицію таймера на екрані, налаштувати формат відображення, змінити шрифт та колір тексту. Після завершення налаштувань можна розпочати гру.

Під час проходження гри таймер оновлюється в реальному часі, фіксуючи всі ключові події відповідно до категорії спідрану. Користувач має можливість паузи, перезапуску або зміни параметрів таймера у відповідному меню модифікації.

Завершення проходження визначається за умовами вибраної категорії (наприклад, перемога над Драконом Краю або досягнення певної події). Після завершення спідрану система генерує детальний звіт, що містить загальний час проходження, проміжні спліти, кількість перезапусків та інші параметри. Також доступна можливість експорту даних у формати .csv або .json для подальшого аналізу чи подачі на speedrun.com.

Для глибшого розуміння логіки функціонування основних компонентів програмного засобу Speedrun Timer було створено покроковий алгоритм, що описує процес взаємодії між модулями системи. Така схема дозволяє не лише відобразити загальну структуру функціонування програми, але й детально

проаналізувати її поведінку на кожному з ключових етапів – від ініціалізації й налаштування до відстеження проходження та завершення ігрової сесії. Особливу увагу в алгоритмі приділено перевірці цілісності моду, зчитуванню вхідних параметрів, застосуванню користувацьких налаштувань таймера, а також ініціалізації категорій спідрану, що є критичними для забезпечення точного обліку часу.

Детальний структурний аналіз допомагає зрозуміти, які саме внутрішні операції виконує кожна з функцій системи, яким чином програма реагує на події в ігровому середовищі та як забезпечується узгодження з конфігураційними даними користувача. Візуалізовану структуру алгоритму роботи системи наглядно показано на рисунку 2.8.

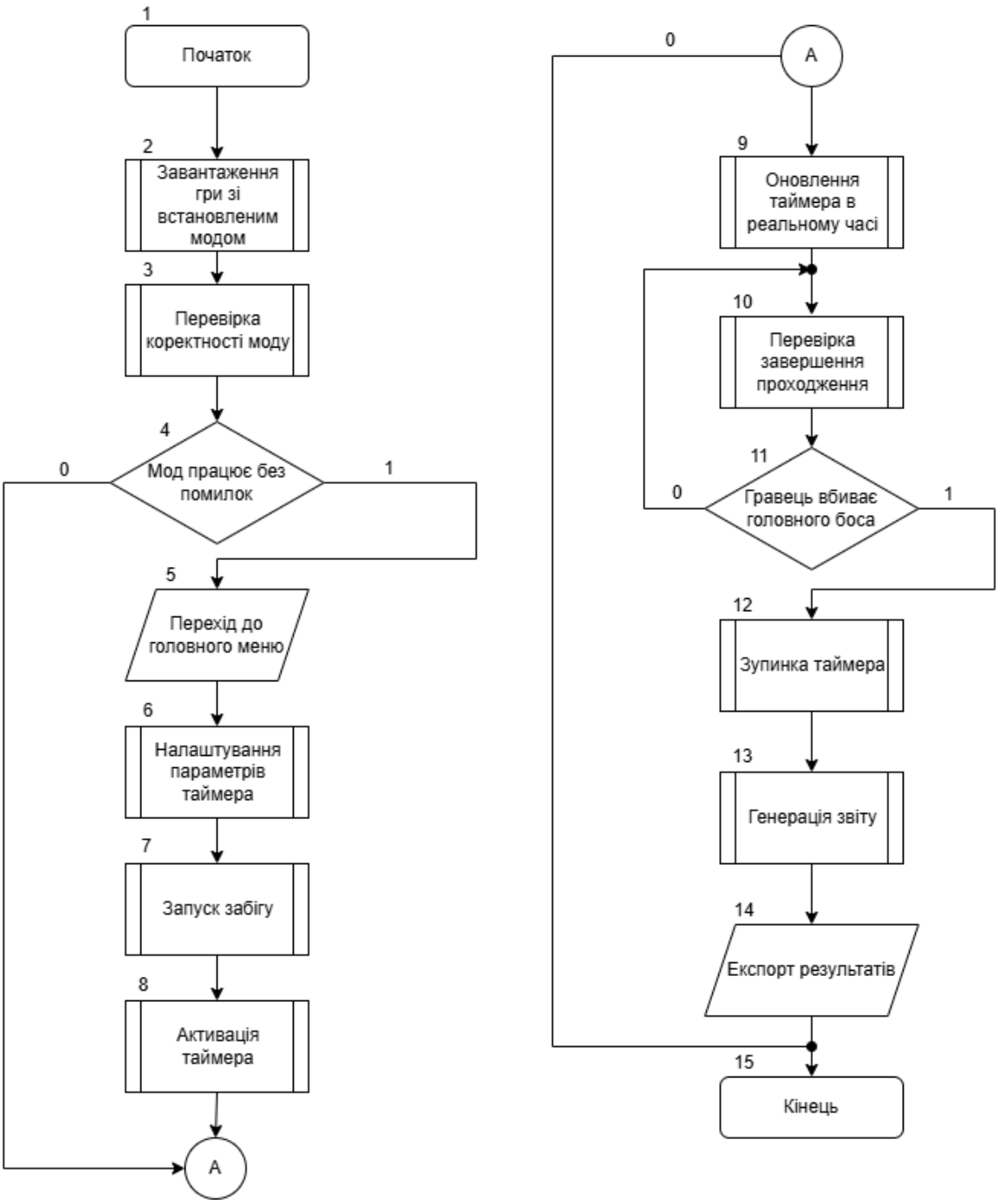


Рисунок 2.8 – Алгоритм функціонування системи

Окрім загального алгоритму роботи, важливою складовою функціональності системи є механізм збору та обробки розширеної статистики спроб користувача. Статистичний модуль фіксує точний час на кожному контрольному етапі та зберігає інформацію про ефективність проходження. Завдяки цьому компонент набуває статусу не просто допоміжного, а одного з ключових засобів аналізу ігрової продуктивності.

Серед додаткових можливостей модуля – перевірка цілісності отриманих даних та підтримка експорту в універсальні формати, зручні для зовнішнього аналізу. Це надає змогу гравцям глибше дослідити власні результати, оцінити динаміку зростання навичок, а також представити ці дані на офіційних ресурсах. Алгоритм функціонування модуля статистики подано на рисунку 2.9.

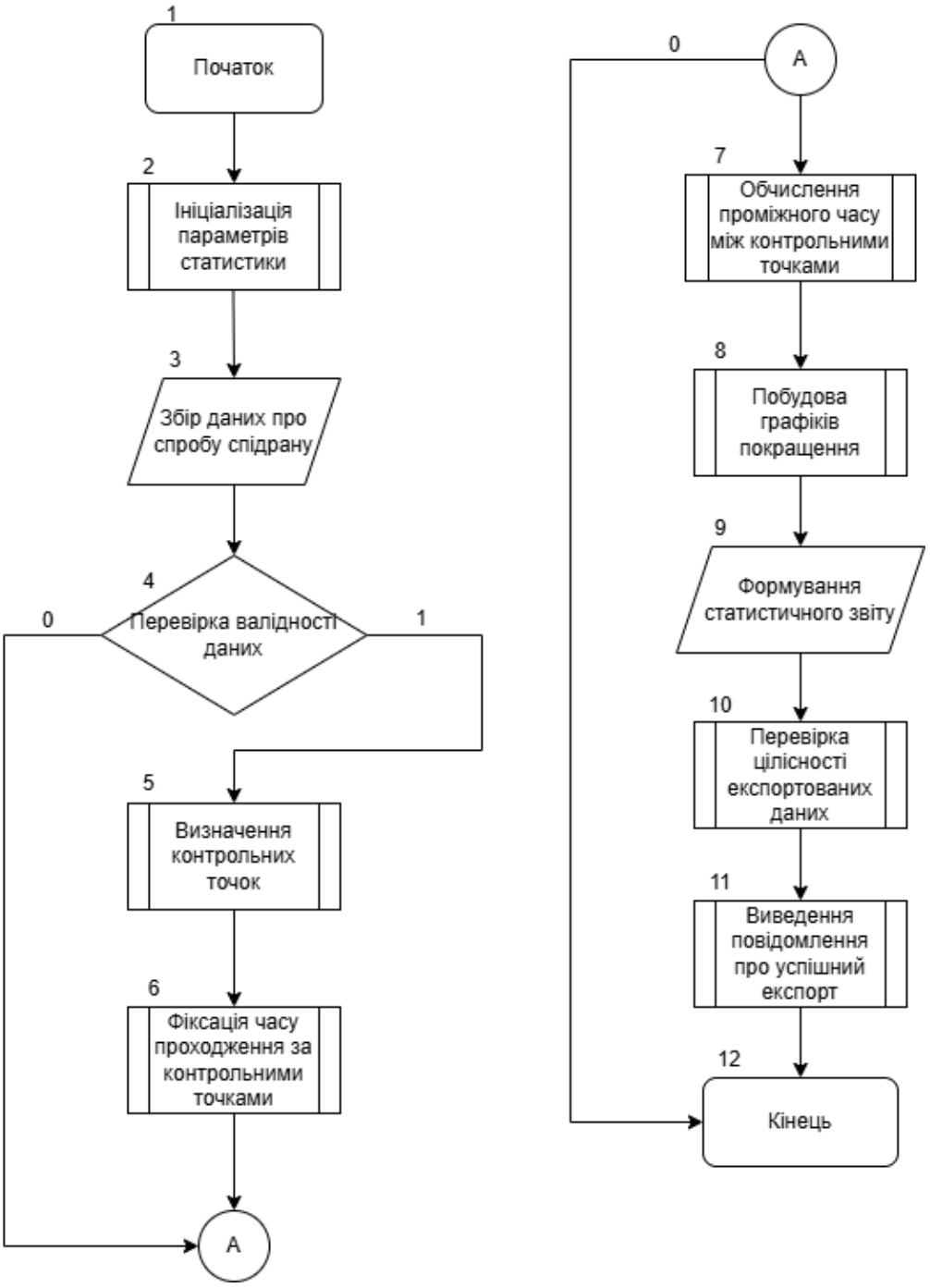


Рисунок 2.9 – Алгоритм збору та відображення статистики спідранів

Для більш глибокого розуміння роботи підсистем, зокрема ядра таймера, яке виконує точний облік часу, було розроблено окрему схему, що докладно ілюструє логіку обчислення проміжків часу між подіями. Цей модуль відповідає за запуск, оновлення, зупинку та зберігання значень відліку часу, і є центральним елементом функціонування всього програмного забезпечення. Його коректна реалізація забезпечує точність аналітичних даних та ефективність застосунку в цілому.

Схема демонструє повну послідовність дій – від моменту активації до фіксації кожної одиниці часу проходження, з урахуванням станів гри, затримок, пауз або перезапусків. Це дозволяє як розробникам, так і користувачам, краще зрозуміти внутрішню логіку системи, а також оцінити її надійність. Умови запуску та зупинки таймера враховують як події в самій грі, так і обрані параметри категорії спідрану, що робить модуль універсальним інструментом для різних сценаріїв використання. Схему алгоритму дії модуля відліку часу наведено на рисунку 2.10.

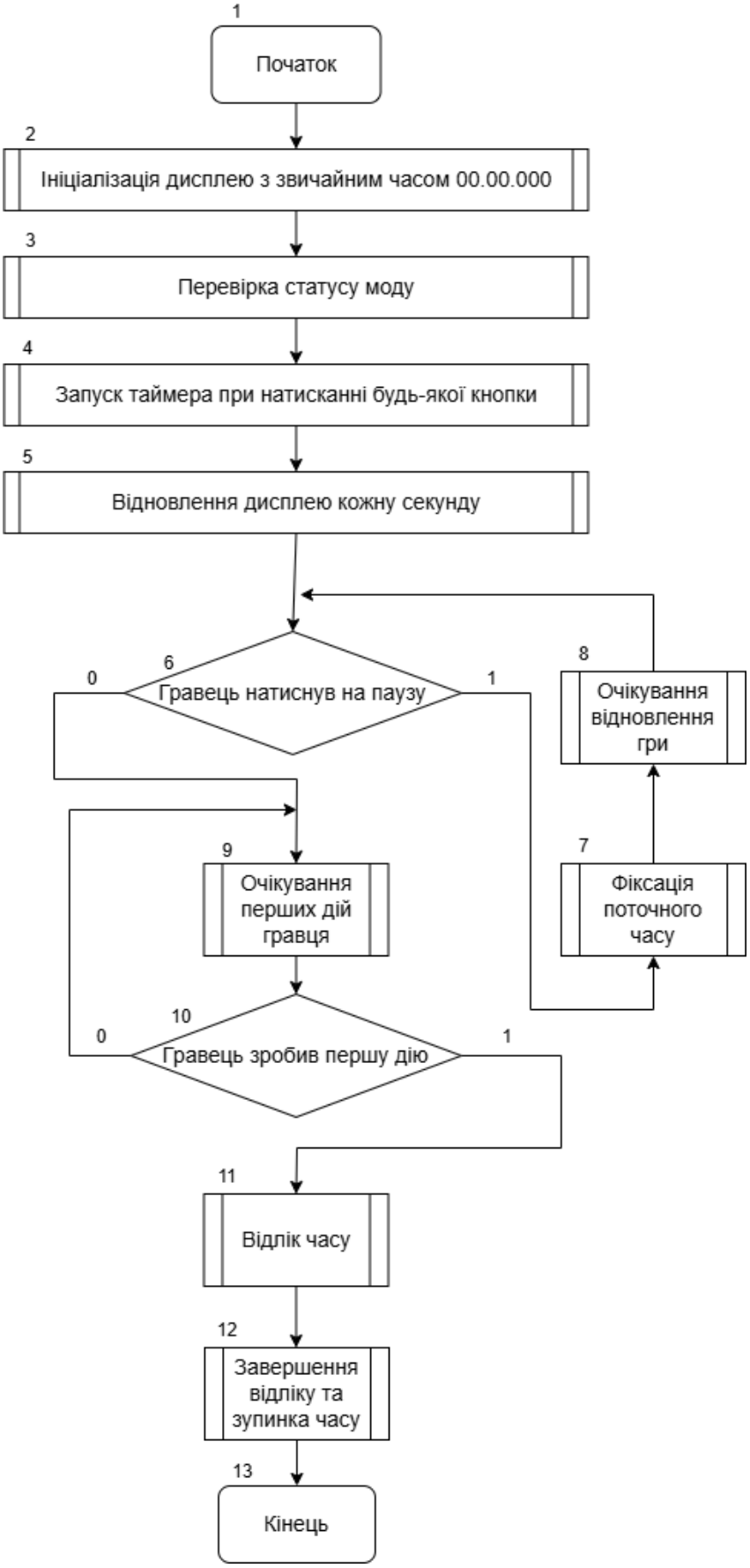

1
Початок
2
Ініціалізація дисплею з звичайним часом 00.00.000
3
Перевірка статусу моду
4
Запуск таймера при натисканні будь-якої кнопки
5
Відновлення дисплею кожну секунду
6
Гравець натиснув на паузу
0
1
8
Очікування відновлення гри
7
Фіксація поточного часу
9
Очікування перших дій гравця
10
Гравець зробив першу дію
0
1
11
Відлік часу
12
Завершення відліку та зупинка часу
13
Кінець

Рисунок 2.10 – Алгоритм роботи модуля відліку часу

Розробка алгоритму роботи програми та інших компонентів системи були виконані за допомогою онлайн сервісу створення схем та діаграм Draw.io [18].

2.4 Розробка системи модульної інтеграції

Система модульної інтеграції, розроблена в межах програмного засобу Speedrun Timer, забезпечує злагоджену взаємодію між його основними компонентами, дозволяючи досягти високої гнучкості, розширюваності та стабільності. Її основне завдання полягає в організації зв’язку між ядром таймера, модулями обробки подій, графічним інтерфейсом, конфігураційною підсистемою та блоками збереження статистики користувача. Такий підхід дозволяє ізолювати логіку кожного модуля, що, у свою чергу, спрощує оновлення та масштабування системи без ризику порушити цілісність загального функціоналу.

Архітектура системи побудована на принципах інверсії залежностей. Кожен функціональний блок реалізовано як незалежний модуль, що взаємодіє з іншими через інтерфейси та централізований диспетчер – головний контролер (MainController). Це дозволяє легко змінювати конфігурацію взаємодій без необхідності модифікації існуючого коду.

Основу взаємодії між компонентами становить подійна модель, реалізована за допомогою Event Bus Fabric API. Внутрішньоігрові події – такі як запуск світу, переміщення гравця між вимірами, досягнення фінальної мети – фіксуються обробником подій (Event Processor), після чого передаються іншим модулям через відповідні канали обробки. Завдяки цьому можлива реакція системи в реальному часі без затримок і надлишкової перевірки умов.

Модуль конфігурації (User Config Module) зчитує та зберігає налаштування користувача у форматі JSON або YAML за допомогою Config API. Налаштування визначають активні модулі (наприклад, ручний чи

автоматичний старт), стиль інтерфейсу та рівень деталізації статистики. Якщо активовано автоматичний запуск, ініціалізується модуль StartTrigger, тоді як у ручному режимі цей компонент залишається неактивним.

Модуль збору статистики (Statistics Module) відповідає за реєстрацію сплітів, створення об’єктів проходження (RunInfo) та збереження їх через File Exporter у форматах JSON/CSV. Інтерфейс виводиться через Overlay Renderer, який відображає таймер на екрані гри з урахуванням вибраних параметрів стилізації.

Діаграму компонентів системи, що ілюструє взаємозв’язки між модулями, представлено на рисунку 2.11.

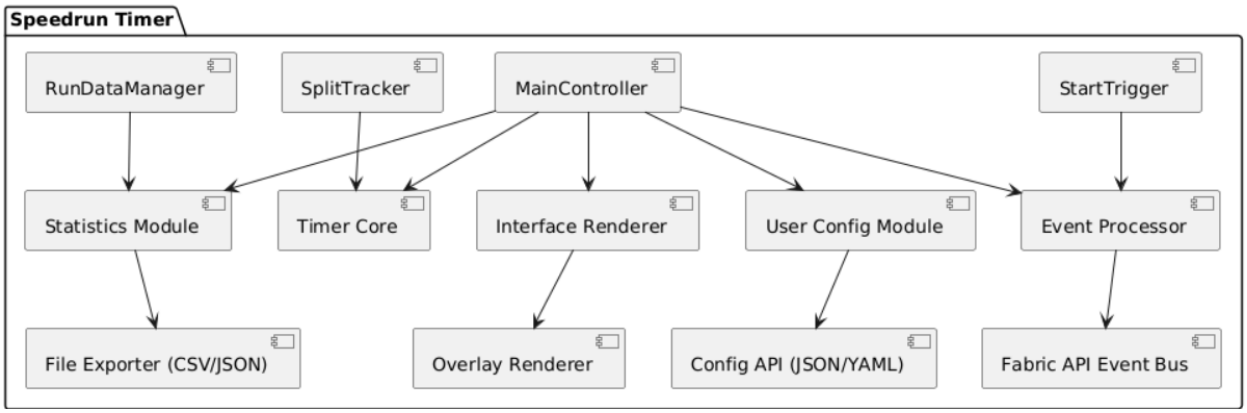


Рисунок 2.11 – Діаграма компонентів системи модульної інтеграції

На діаграмі відображено основні компоненти: центральний контролер (MainController), ядро таймера (Timer Core), обробник подій (Event Processor), модулі конфігурації, статистики та інтерфейсу. Зв’язки між модулями реалізовано через Fabric API Event Bus, а дані зберігаються за допомогою Config API та File Exporter. Така структура забезпечує адаптивність, гнучкість та можливість подальшого масштабування функціональності модуля без втрати стабільності.

Завдяки впровадженню модульної архітектури стало можливим не лише організувати ефективну взаємодію частин системи, а й забезпечити її адаптацію

до змін середовища Minecraft та вимог користувачів без необхідності глибоких переробок. Це особливо важливо для підтримки модифікації в актуальному стані при оновленнях гри та додаванні нових функцій.

2.5 Висновки

У другому розділі було проведено аналіз функціональних можливостей та особливостей роботи моду Speedrun Timer. Також було розглянуто процес налаштування параметрів таймера та інтеграцію з грою Minecraft. Було розроблено основний алгоритм роботи програмного засобу, що включає запуск, контроль ключових подій та завершення проходження. Крім того, було створено схеми зберігання результатів та відліку часу таймера.

3 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

3.1 Обґрунтування вибору програмних засобів для розробки

Розробка внутрішньоігрового таймера для спідранів у середовищі Minecraft потребує ретельного вибору інструментів, мов програмування та архітектурних підходів. Оскільки йдеться про модифікацію гри з глибокою інтеграцією у її внутрішні події, основними критеріями стали: сумісність з останніми версіями Minecraft, продуктивність, гнучкість, наявність підтримки спільнотою, розширюваність, а також мінімізація втручання у базовий код гри.

Основу набору інструментів розробки склала платформа Fabric API. Цей фреймворк зарекомендував себе як один із найефективніших засобів для створення модифікацій Minecraft, зокрема у тих випадках, коли важлива висока продуктивність і стабільність при одночасній роботі з іншими модами. Fabric API дозволяє розробникам реалізовувати новий функціонал, не змінюючи вихідний код гри, а лише доповнюючи його. Завдяки модульній архітектурі та системі інжекцій, Fabric підтримує швидку адаптацію до нових версій Minecraft, що є критично важливим у постійно оновлюваному середовищі гри.

Мова програмування Java була обрана як основний інструмент розробки, оскільки саме на ній побудовано ядро Minecraft. Це забезпечує повну інтеграцію з ігровими об’єктами, подіями, системами координат, інвентарем, обробкою сутностей та іншими важливими аспектами ігрового світу. Окрім того, Java має розвинену систему бібліотек, потужну підтримку розробників та гнучку структуру об’єктно-орієнтованого програмування, що полегшує реалізацію складних функціональних блоків, таких як обробники подій, система відліку часу, експорт статистики, підтримка користувацьких категорій.

Особливе значення при розробці має технологія Mixin [19], яка дозволяє модифікувати наявний ігровий код без необхідності його прямого переписування. Mixin функціонує як рівень інжекцій, що вставляє власні методи у вже скомпільовані класи Minecraft. Це надзвичайно важливо для збереження сумісності з іншими модифікаціями, а також для підтримки

стабільності роботи гри в цілому. За допомогою Mixin реалізовано перехоплення внутрішньоігрових подій, запуску гри, зміни фаз, а також реєстрація тригерів сплітів, що унеможливлюється у разі використання класичних жорстко прописаних методів.

Іншою невід'ємною складовою є Minecraft Tick System [20], що становить фундаментальний механізм для оновлення стану світу гри. Minecraft працює на основі тикової системи, де кожна дія фіксується з інтервалом у 1/20 секунди. Таймер Speedrun Timer прив'язано саме до цієї системи, що дозволяє реалізувати надзвичайно точний відлік часу без затримок і без потреби в сторонніх таймерах. Вимірювання часу ґрунтується на підрахунку кількості ігрових тіків між подіями, що гарантує відповідність ігровій дійсності та точність з погляду спідрану.

Для збереження та обробки налаштувань користувача, категорій проходження та станів системи було використано Config API [21] – стандартний засіб створення конфігурацій у середовищі Fabric. Цей API дозволяє зберігати всі параметри у форматах JSON/YAML, забезпечуючи зручну структуру, можливість локального редагування, переносу між пристроями, а також підтримку різних конфігурацій для різних сценаріїв гри. Завдяки цьому таймер можна швидко адаптувати під будь-які потреби гравця: від звичайного проходження до хардкорного режиму з індивідуальними умовами.

Крім того, у структурі розробки передбачено використання таких технологій, як Event Bus Fabric [22], що забезпечує підписку на ігрові події у реальному часі, а також бібліотеки рендерингу інтерфейсу для виведення таймера на екран без втручання у механіку самої гри. Це дає змогу створити динамічний, легкий і кастомізований інтерфейс, який не перевантажує гру навіть за умов використання численних модів.

Поєднання Fabric API, Java, Mixin, Minecraft Tick System, Config API та супровідних технологій дозволяє досягти високої гнучкості та масштабованості продукту. Водночас це забезпечує точне позиціонування таймера в контексті ігрової логіки Minecraft, відповідність вимогам спідранерської спільноти,

підтримку статистики та кастомних категорій. Отже, вибрані інструменти дозволяють реалізувати стабільний, точний і зручний таймер для спідрану, який повністю інтегрується у внутрішні механізми Minecraft. Завдяки цьому таймер може працювати синхронно з ігровими подіями та легко адаптується до різних версій гри, що робить його надійним рішенням для спільноти гравців.

3.2 Вибір середовища розробки

Одним із ключових етапів у створенні модифікації Speedrun Timer стало визначення найбільш ефективного середовища розробки. Оскільки мод створюється для гри Minecraft із використанням Fabric API, середовище мало відповідати ряду технічних вимог: підтримка мови Java, сумісність із Gradle [23], можливість відлагодження, стабільність при тривалому використанні, підтримка складних модульних структур та інтеграція з ігровим клієнтом Minecraft.

У результаті порівняльного аналізу середовищ було обрано середовище IntelliJ IDEA [24] – одну з найпопулярніших та найпотужніших інтегрованих середовищ розробки для мови Java. Її переваги включають глибоку інтеграцію з Gradle-проєктами, розширені можливості налагодження коду, гнучке автодоповнення, підтримку Git та розширень, а також повну сумісність із проєктною структурою Minecraft-модів, побудованих на Fabric.

Використання IntelliJ IDEA забезпечує стабільну роботу з такими конфігураційними файлами, як build.gradle, settings.gradle, fabric.mod.json, що відіграють ключову роль у конфігурації моду. Середовище дозволяє безперешкодно додавати зовнішні залежності, серед яких – Fabric API, Cloth Config API та Event Bus Fabric, що імпортуються автоматично через систему керування залежностями Gradle.

Особливо зручним є вбудований механізм запуску Minecraft з підтримкою режиму відлагодження (debug mode), що дозволяє запускати гру безпосередньо з IntelliJ IDEA, із можливістю встановлення точок зупинки, перегляду змінних, трасування викликів методів та покрокового аналізу логіки. Це критично

важливо при розробці компонентів таймера, які взаємодіють із внутрішніми подіями Minecraft у реальному часі.

IntelliJ IDEA підтримує сучасні стандарти Java, зокрема роботу з лямбда-виразами, потоками, сучасними структурами даних, що суттєво спрощує реалізацію обробки подій гри, збереження даних користувача та генерацію результатів спідрану. Додатково, завдяки інтеграції з Git, стало можливим вести контроль версій, створювати окремі гілки розробки для експериментальних функцій, а також зберігати історію змін.

Середовище забезпечує зручну навігацію між модулями, має потужну систему пошуку, автоматичну генерацію шаблонів коду, а також розширюваність через плагіни, наприклад для роботи з JSON, YAML або аналізу структури класів Minecraft. Завдяки цьому розробка ведеться швидко, структуровано та із мінімальними ризиками помилок.

Отже, обрання IntelliJ IDEA як основного середовища розробки для створення модифікації Speedrun Timer є технічно обґрунтованим кроком, який дозволяє реалізувати повноцінний життєвий цикл проєкту – від створення до тестування, збирання та експорту стабільного .jar-файлу, готового до використання в Minecraft.

3.3 Інструкція користувача

Інструкція користувача програмного модуля Speedrun Timer містить основні положення, що стосуються встановлення, налаштування та використання таймера в середовищі гри Minecraft. Вона орієнтована на забезпечення зручного і зрозумілого процесу взаємодії з модифікацією, а також сприяє ефективному використанню її функціональних можливостей під час проходження гри у форматі спідрану.

На етапі первинної ініціалізації роботи з програмним модулем необхідно враховувати мінімальні та рекомендовані технічні характеристики персонального комп’ютера, що є критично важливим для забезпечення стабільної та безперебійної роботи як модифікації, так і гри в цілому.

Розроблений таймер є інтегрованим доповненням до гри Minecraft та функціонує на базі модифікатора Fabric відповідної версії. У таблиці 3.1 наведено рекомендовані та мінімальні технічні характеристики для запуску програми, що дозволяють уникнути можливих збоїв під час роботи таймера.

Таблиця 3.1 – Вимоги до апаратного забезпечення

Параметр конфігурації	Рекомендовані значення	Мінімально допустимі значення
Центральний процесор	Intel Core i5 / AMD Ryzen 5	Intel Core i3 / AMD Athlon
Оперативна пам'ять (RAM)	8 ГБ	4 ГБ
Обсяг вільного дискового простору	10 ГБ	3 ГБ
Графічний адаптер	NVIDIA GeForce GTX 1050 Ti або еквівалент	Intel HD Graphics або еквівалент
Операційна система	Windows 10 / 11, Linux, macOS	Windows 7
Додаткове ПЗ	Встановлений Minecraft з Fabric Loader	Minecraft версії 1.19+

Наступним етапом є інтеграція файлів програмного модуля Speedrun Timer до ігрового клієнта Minecraft. Для цього необхідно здійснити копіювання файлу модуля, назва якого відображає його версію (speedrun-timer-1.16.1.jar), до спеціалізованої директорії «mods», що розташовується у кореневому каталозі встановленої гри Minecraft. Візуальне представлення структури файлів після копіювання модуля наведено на рисунку 3.1.

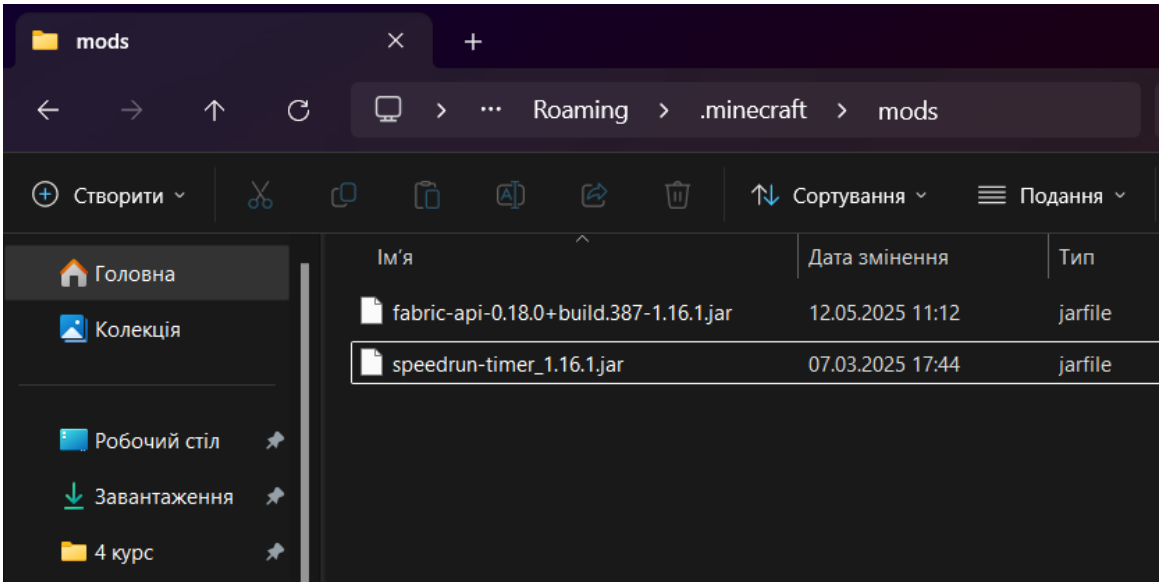


Рисунок 3.1 – Розположення файлу моду

Після розміщення файлу модуля у відповідній директорії, користувачеві необхідно запустити ігровий клієнт Minecraft. У головному меню гри відображається перелік встановлених модифікацій, серед яких має бути присутній Speedrun Timer та Fabric Loader. Наявність модуля у даному списку є підтвердженням його успішної інтеграції до ігрового середовища та готовності до подальшої експлуатації.

Для доступу до меню керування програмним модулем Speedrun Timer безпосередньо під час ігрового процесу передбачено використання гарячої клавіші «T», яка встановлена за замовчуванням. Слід зазначити, що користувач має можливість перепризначення даної клавіші на більш зручну комбінацію відповідно до власних уподобань через відповідні налаштування модуля. Меню керування надає користувачеві інтуїтивно зрозумілий інтерфейс для конфігурування різноманітних параметрів роботи таймера.

Основні функціональні можливості модуля включають вибір категорії спідрану, що забезпечує автоматичну фіксацію релевантних ігрових подій; активацію автоматичного запуску таймера при виконанні першої значущої

ігрової дії; а також налаштування візуального відображення таймера на екрані, включаючи зміну шрифту, кольору та положення. Візуалізація інтерфейсу меню керування програмним модулем Speedrun Timer представлена на рисунку 3.2.

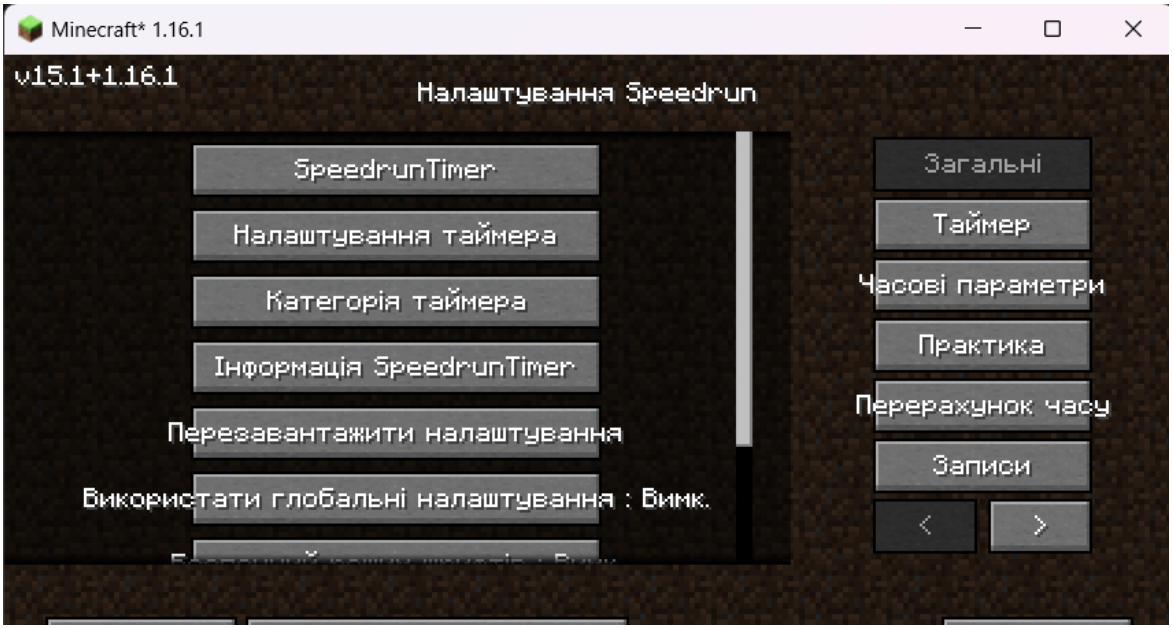


Рисунок 3.2 – Меню керування таймера

Після здійснення необхідних налаштувань, гравець може розпочати проходження гри. Залежно від обраного режиму запуску, активація таймера відбудеться автоматично або вручну за допомогою призначеної гарячої клавіші. Протягом ігрового процесу модуль Speedrun Timer здійснює безперервний моніторинг та фіксацію ключових ігрових подій, перелік яких варіюється залежно від обраної категорії спідрану. До таких подій належать входження до Нижнього світу, перехід до Краю, смерть персонажа, а також успішне завершення гри шляхом перемоги над Драконом Краю. Після досягнення умов завершення спідрану для обраної категорії, таймер автоматично зупиняє відлік часу, надаючи користувачеві можливість збереження отриманих результатів.

Після завершення ігрової сесії спідрану на екрані користувача відображається підсумкове вікно, що містить детальну інформацію про

зафіксований забіг. До основних відображуваних параметрів належать загальний час проходження гри, проміжні часові відмітки для ключових ігрових подій (спліти), загальна кількість спроб проходження для поточної категорії, а також середнє значення частоти кадрів (FPS) протягом усієї ігрової сесії. У нижній частині екрана розташована кнопка для експорту отриманих результатів у форматах CSV або JSON. Дана функціональність забезпечує можливість подальшого аналізу, обробки та публікації результатів на спеціалізованих платформах для спідранерів.

3.4 Програмна реалізація

Архітектурне проєктування таймера реалізовано як багатокомпонентну систему. Реалізація базується на модульному підході: окремі класи відповідають за облік часу, реакцію на ігрові події, обробку даних користувача, рендеринг інтерфейсу, конфігурацію та збереження результатів. Така побудова забезпечує стабільність роботи, гнучке масштабування та полегшує підтримку моду в нових версіях Minecraft.

Основним класом є Timer, що виступає обчислювальним ядром системи. Він містить механізми запуску й зупинки таймера, обчислення тривалості сесії та зберігання поточного стану. Облік часу відбувається через TickHandler, який реєструє кожен ігровий тік. Клас Timer також взаємодіє з SplitTracker, відповідальним за фіксацію сегментів проходження. Оскільки дані можуть оновлюватися асинхронно, реалізовано механізм буферизації зміни станів. З метою глибшого осягнення структури представленого класу було розроблено відповідну діаграму, яку наведено на рисунку 3.3.

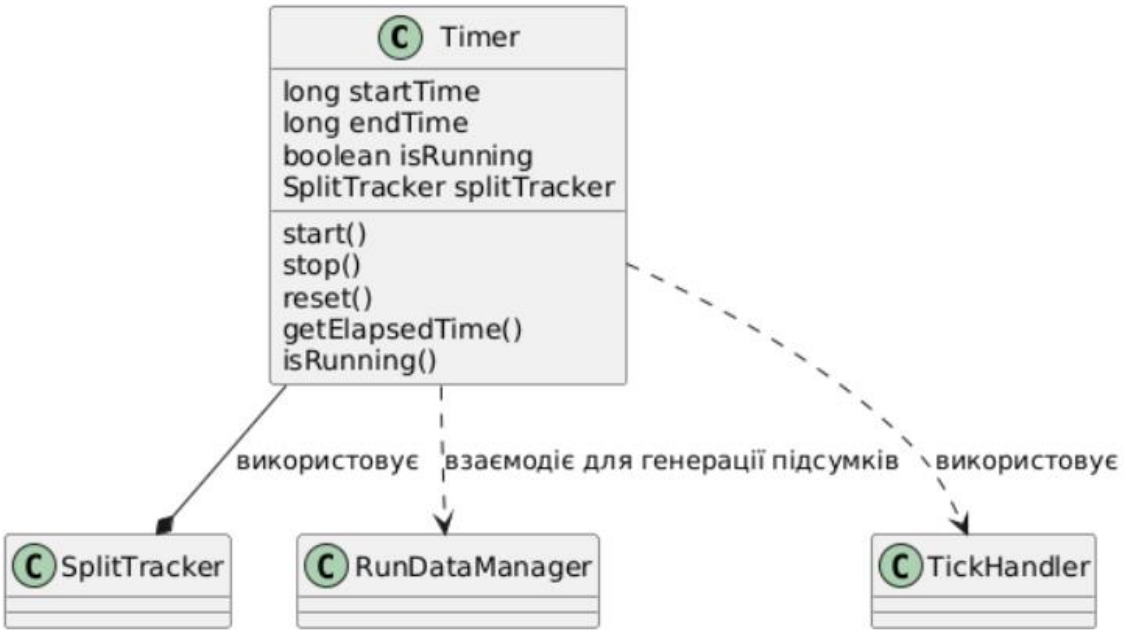


Рисунок 3.3 – Діаграма класу Timer

Реакція на події гри делегується класу GameHooks. Він відстежує зміни вимірів, старт світів, завершення гри (перемога над драконом, смерть), а також сигнали першого введення (First Input), які активують старт таймера. Для цього застосовуються стандартні події Fabric API, що надходять через EventListenerRegistry. Також підключено обробку raw-даних, що надходять безпосередньо від клієнта Minecraft – для підвищення точності. Візуалізацію архітектури розглядуваного класу з метою покращення її сприйняття відображено на діаграмі, що ілюструється рисунком 3.4.

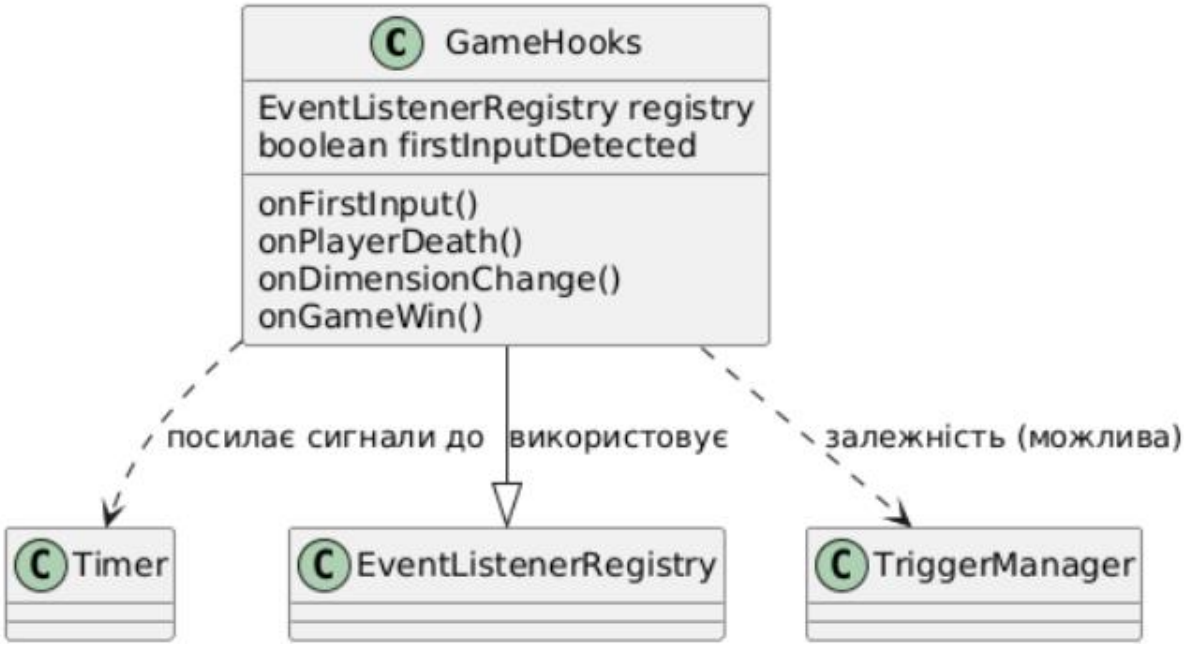


Рисунок 3.4 – Діаграма класу GameHooks

Графічний інтерфейс реалізовано у вигляді класу OverlayRenderer, що малює таймер поверх гри. Цей клас використовує методи Minecraft RenderSystem і підтримує позиціювання, прозорість, стилізацію шрифтів. Стани сегментів, активність, пауза, завершення та колірна схема адаптуються до вибраного профілю. З метою забезпечення наочного представлення функціональних зв'язків даного класу було сформовано діаграму, ознайомитися з якою можна на рисунку 3.5.

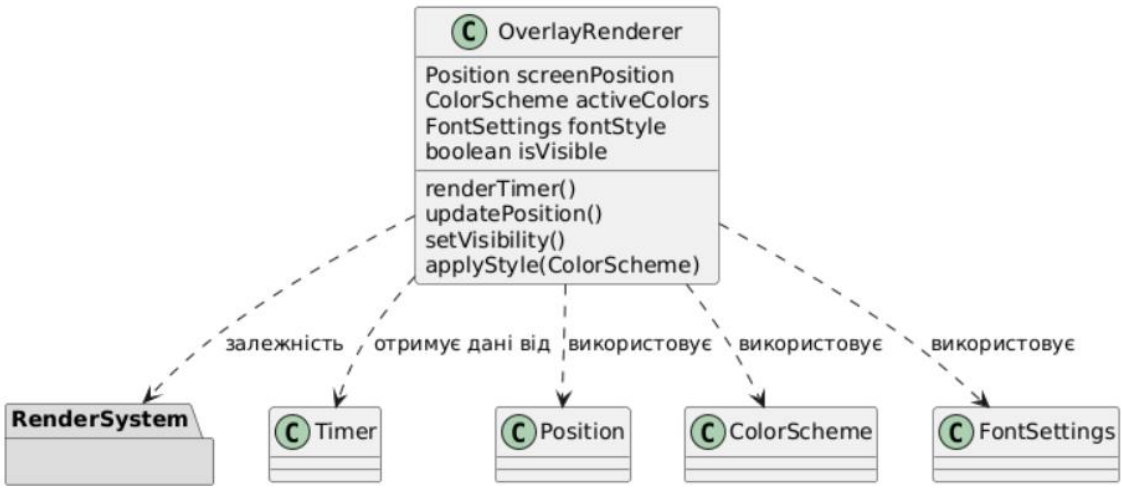


Рисунок 3.5 – Діаграма класу OverlayRenderer

Файл конфігурації обробляється через клас ModConfig. У ньому реалізовано серіалізацію налаштувань користувача у форматі JSON – вибір категорії спідрану, стилю інтерфейсу, типу запуску (авто/ручний), активність сплітів, статистику. Зміни у конфігурації застосовуються в реальному часі, без перезапуску гри. Для графічного відображення меню налаштувань використано Cloth Config API. Для кращого розуміння було вирішено створити діаграму класу, показану на рисунку 3.6.

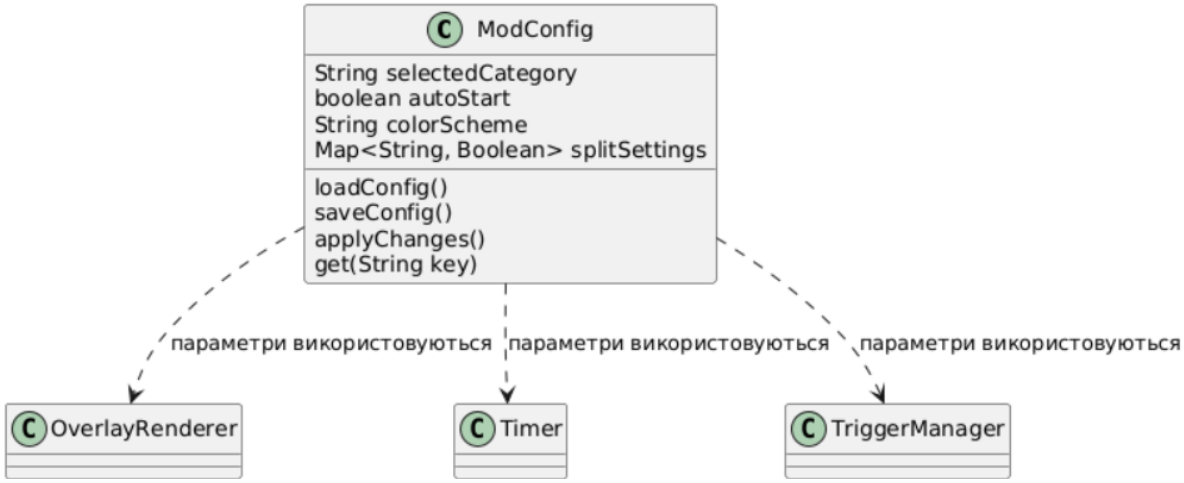


Рисунок 3.6 – Діаграма класу ModConfig

Після завершення гри RunDataManager генерує об’єкт RunInfo, що містить усі метадані про проходження: тривалість, кількість сплітів, координати спрацювання подій, час кожного сегмента, FPS. Ці дані експортуються у форматах .json і .csv з опціональним зазначенням директорії користувача.

У процесі обробки внутрішньоігрових подій, зокрема виявлення ключових моментів проходження та синхронізації станів гри, було застосовано Minecraft Mapping Tools (MMT). Їх інтеграція дозволила підвищити точність

визначення подій та адаптувати логіку таймера до конкретного контексту ігрового середовища.

Запис дій реалізовано у класі LoggerWrapper, що базується на java.util.logging.Logger. Він забезпечує ведення внутрішніх логів: старт таймера, перехід між фазами, помилки у виконанні. Формат логів адаптовано для зручного перегляду та аналізу. При потребі звіти можуть бути використані для діагностики або подання у спільнотах speedrun.com.

3.5 Висновки

У цьому розділі детально розглянуто розробку Speedrun Timer для Minecraft. Обґрунтовано вибір Fabric API, Java та Mixin як ключових інструментів для інтеграції, продуктивності та сумісності. Підкреслено важливість Tick System для точності таймера та Config API для керування налаштуваннями.

Вибір IntelliJ IDEA як IDE забезпечив ефективну розробку завдяки налагодженню та підтримці Fabric. Надано інструкцію користувача для встановлення та використання. Програмна реалізація має модульну архітектуру (класи Timer, GameHooks, OverlayRenderer, ModConfig), що забезпечує стабільність та гнучкість. UML-діаграми візуалізують структуру класів. Описані технології дозволили створити інтегрований, точний та зручний інструмент для спідранерів Minecraft.

4 ТЕСТУВАННЯ ПРОГРАМИ

4.1 Вибір методики тестування

Для перевірки працездатності, точності та стабільності програмного модуля Speedrun Timer була обрана поетапна методика тестування, що передбачає як функціональні, так і нефункціональні випробування. Методика тестування ґрунтується на загальноприйнятих принципах забезпечення якості програмного забезпечення та адаптована під специфіку ігрового середовища Minecraft.

На першому етапі проводиться функціональне тестування – воно дозволяє перевірити реалізацію основної логіки моду: запуск, зупинка та скидання таймера, взаємодія через графічний інтерфейс, а також реагування на гарячі клавіші. Цей етап дає змогу виявити помилки в реалізації базових функцій.

Наступним етапом є тестування надійності, що полягає у перевірці поведінки моду в умовах нестабільного ігрового середовища або підвищеного навантаження. Особливу увагу приділяється стійкості до раптового завершення гри, виходу користувача, перевантаження обчислень та частих змін параметрів конфігурації.

Далі здійснюється тестування продуктивності, метою якого є визначення впливу моду на ігровий FPS, використання пам’яті та навантаження на процесор. Особлива увага приділяється оптимізації графічного виводу таймера та реакції моду в складних ігрових сценаріях.

Завершальним кроком є перевірка сумісності з іншими модами. Це дозволяє виявити потенційні конфлікти у взаємодії з популярними модами типу MiniHUD [24], Sodium [25] або Lithium [26], які часто використовуються спільнотою спідранерів. Така перевірка є необхідною для забезпечення максимальної універсальності таймера.

4.2 Тестування програмного забезпечення

Тестування програмного модуля Speedrun Timer проводилось з метою перевірки відповідності функціоналу заявленим вимогам, виявлення можливих помилок, оцінки стабільності та впливу моду на ігрове середовище Minecraft. Основною метою тестування було забезпечення коректної роботи таймера у реальних умовах використання гравцями-спідранерами.

Перед початком тестування було сформовано спеціальне тестове середовище: встановлено відповідну версію Minecraft з підтримкою Fabric Loader, створено окремий ігровий світ, а також підключено допоміжні модифікації (MiniHUD, Sodium) для перевірки сумісності. Додатково налаштовано конфігураційні параметри таймера для відстеження різних сценаріїв його запуску та використання.

На першому етапі здійснено функціональне тестування, яке охоплювало базові дії користувача: запуск, зупинка та скидання таймера. Перевірено також роботу гарячих клавіш та GUI-інтерфейсу. Результати засвідчили правильне відображення часу, стабільну реакцію на взаємодію з користувачем та коректне оновлення відліку часу у режимі реального часу.

Другим етапом стало тестування надійності, де мод перевірявся у ситуаціях з підвищеним навантаженням: швидка зміна налаштувань, повторний запуск, багатократне перемикання режимів. Особливу увагу приділено реакції на вихід з гри та раптове завершення процесу Minecraft. У всіх випадках таймер демонстрував стійкість до зовнішніх змін і зберігав останній зафіксований стан. (див. рисунок 4.1).

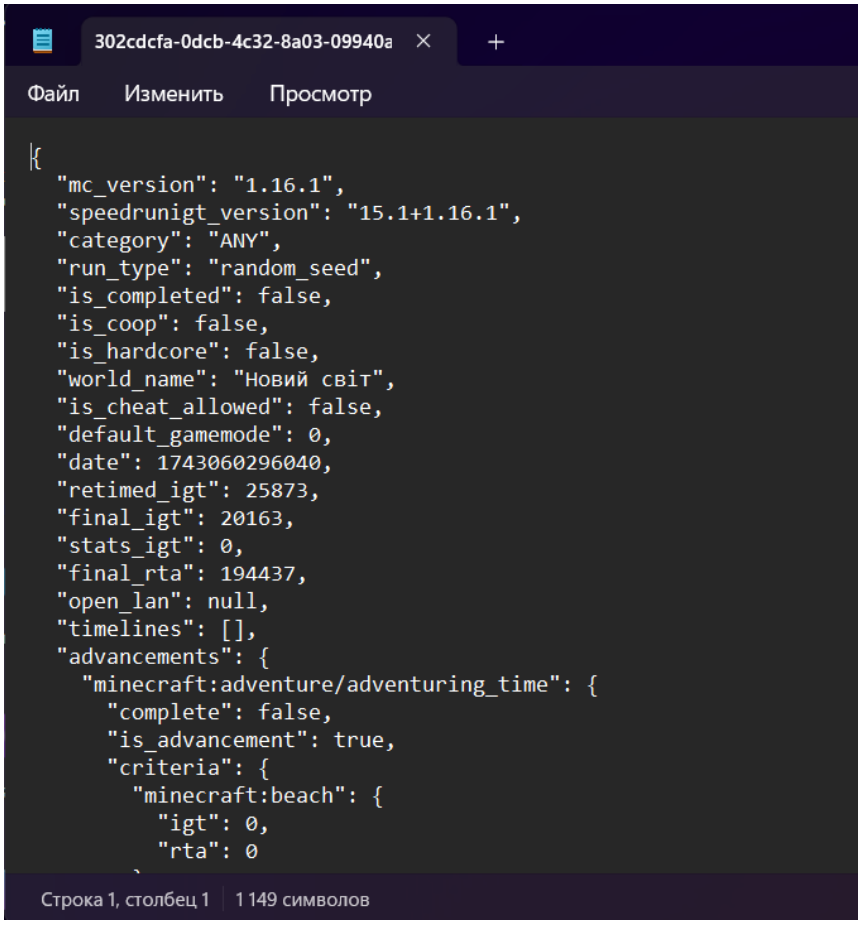


Рисунок 4.1 – Результат проходження після раптового завершення гри

Наступним етапом було тестування продуктивності. Вимірювались показники FPS, використання оперативної пам’яті та процесора під час активної роботи моду. За результатами тестів, таймер не спричиняв помітного падіння продуктивності, не створював надмірного навантаження та не конфліктував із рендерингом Minecraft. Це свідчить про ефективну оптимізацію алгоритмів та рендерингових компонентів.

Окрему увагу приділено сумісності моду з іншими модифікаціями. Було протестовано інтеграцію з MiniHUD, Lithium та іншими популярними модами. Мод Speedrun Timer успішно функціонував у всіх протестованих комбінаціях, що підтверджує його стабільність та відсутність конфліктів.

Додатково проведено перевірку коректності відображення формату часу: зміну хвилин, секунд, мілісекунд. Таймер відображав значення точно та без

затримок, що є критично важливим для змагального використання (див. рисунок 4.2).

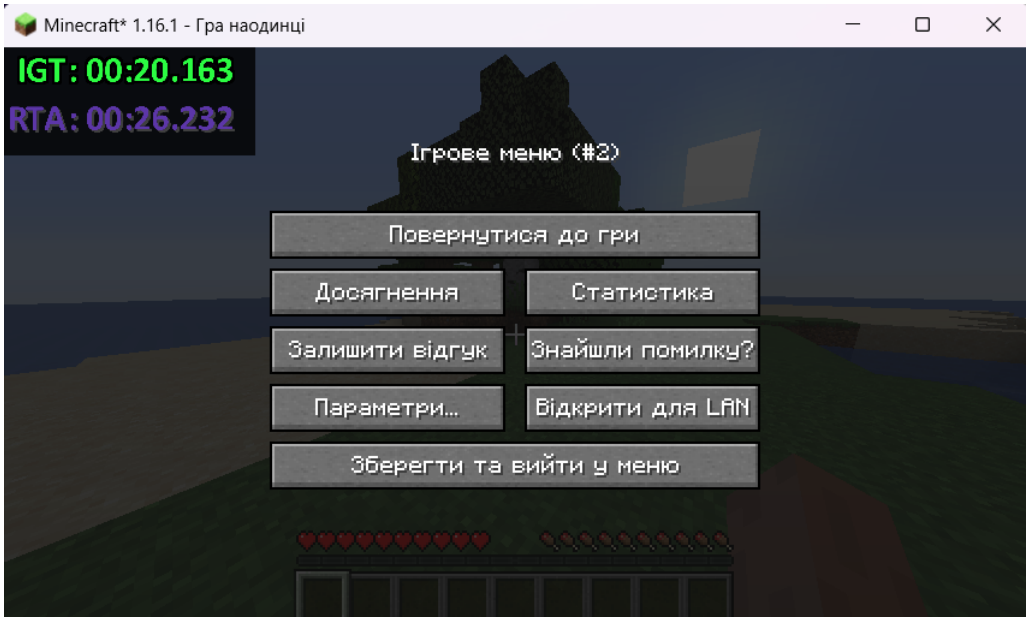


Рисунок 4.2 – Відображення часу проходження

Також було протестовано взаємодію через графічний інтерфейс, де користувач мав змогу керувати модом через елементи GUI. Усі кнопки та дії працювали згідно з очікуваннями, без візуальних артефактів або помилок рендерингу (див. рисунок 4.3).

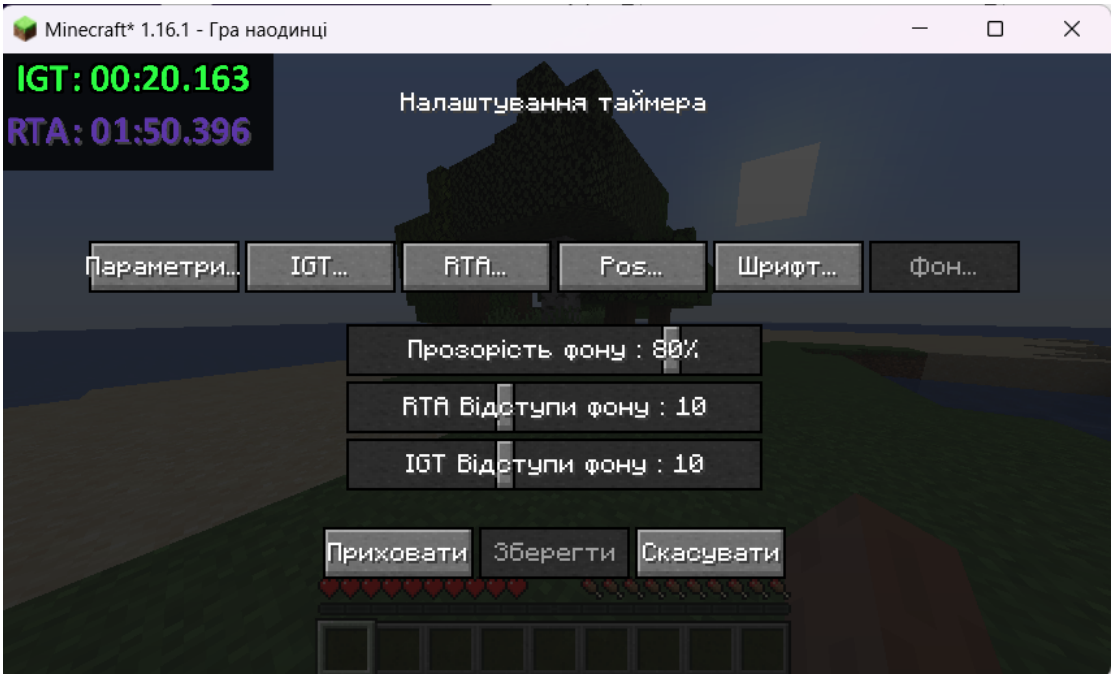


Рисунок 4.3 – Налаштування графічного інтерфейсу

У результаті проведених випробувань було підтверджено, що таймер стабільно функціонує в реальному ігровому середовищі, не викликає критичних помилок, сумісний із поширеними модифікаціями та відповідає вимогам до точності, надійності та зручності користування.

4.3 Висновки

У результаті тестування модуль Speedrun Timer підтвердив відповідність заявленим функціональним вимогам: він стабільно виконує основні операції, має коректно працюючий графічний інтерфейс та зберігає стан навіть у разі виходу з гри чи аварійного завершення процесу. Також було підтверджено відсутність конфліктів із поширеними модами та мінімальний вплив на продуктивність Minecraft. Завдяки гнучкій архітектурі, точному відліку часу та високій сумісності модуль можна ефективно використовувати як для особистого аналізу проходжень, так і для професійного застосування у спільноті спідранерів.

ВИСНОВКИ

У процесі виконання бакалаврської кваліфікаційної роботи було здійснено повноцінне дослідження, реалізацію та тестування програмного рішення, що дозволяє точно фіксувати ігровий час у реальному середовищі Minecraft. Поставлена мета була досягнута шляхом розробки внутрішньоігрового таймера, що здатен автоматично реагувати на ключові події проходження гри, зберігати детальну статистику й забезпечувати користувачеві зручний інтерфейс для налаштування.

У ході реалізації поставлених завдань було виконано такі основні етапи:

- проведено ґрунтовний аналіз сфери спідрану в комп'ютерних іграх та ідентифіковано ключові проблеми, пов'язані з точним вимірюванням часу проходження;
- досліджено існуючі таймери та виокремлено їхні переваги, обмеження, архітектурні особливості та технічні характеристики;
- розроблено власну архітектурну модель, яка ґрунтується на модульному підході та незалежності від зовнішніх засобів фіксації часу;
- обґрунтовано вибір інструментів і технологій, серед яких – Fabric API як базова платформа для створення модифікацій Minecraft, Java як основна мова реалізації, а також Minecraft Mapping Tools (MMT), яка дозволила ефективно працювати зі структурою гри;
- реалізовано ключові програмні модулі: ядро обліку часу (Timer), обробку ігрових подій (GameHooks), графічний рендеринг інтерфейсу (OverlayRenderer), конфігураційний модуль (ModConfig) і блок збору статистики (RunDataManager);
- забезпечено підтримку ручного та автоматичного запуску, кастомізацію зовнішнього вигляду, зберігання результатів у форматах CSV/JSON і відновлення даних після завершення сесії;

- проведено тестування на коректність, стабільність, продуктивність та сумісність із іншими модифікаціями, що підтвердило надійну роботу системи у різних сценаріях використання.

Наукова новизна даної роботи полягає у реалізації концепції повністю інтегрованого внутрішньоігрового таймера, який працює в рамках тікової системи Minecraft і реагує виключно на події, що виникають у грі, без необхідності зовнішнього втручання. Це дозволяє уникнути похибок, зумовлених затримками операційної системи або сторонніх програм. Також запропоновано унікальну структуру модульної інтеграції, що забезпечує легке підключення нових компонентів або сценаріїв використання без необхідності редагування коду інших модулів.

Практичне значення розробленого рішення полягає у можливості його використання реальними спідранерами як на локальному рівні, так і в умовах офіційного проходження із записом результатів. Завдяки підтримці збереження статистики, кастомізації категорій і форматів експорту, модуль є універсальним і легко адаптується до потреб широкого кола користувачів. Його інтеграція в ігрове середовище не викликає конфліктів із базовою логікою Minecraft та забезпечує повну сумісність із популярними модами продуктивності.

Перспективи розвитку проєкту передбачають подальше розширення функціональності таймера: додавання підтримки мультиплеєрних сесій, покращення візуальної частини інтерфейсу, підтримку онлайн-синхронізації результатів із платформами на кшталт speedrun.com, а також оптимізацію для нових версій Minecraft.

Загалом результати, отримані під час розробки та тестування програмного модуля Speedrun Timer, можуть слугувати основою для подальших досліджень у галузі інтегрованих ігрових інструментів та прикладних систем обліку часу в умовах взаємодії з динамічним геймплейним середовищем.

ПЕРЕЛІК ПОСИЛАНЬ

1. Salen, K., Zimmerman, E. Rules of Play: Game Design Fundamentals. – MIT Press, 2004. – 672 p.

2. Войтко В.В., Денисюк А.В., Гаврилюк О.В., Барчук Н.Є., Туровець А. В. Розробка часового відслідковування дій для спідрану у вигляді моду з використанням Fabric API та MMT / Матеріали LIV Всеукраїнської науково-технічної конференції підрозділів Вінницького національного технічного університету: збірник доповідей [Електронний ресурс]: https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/viewFile/25052/20686 (дата звернення: 22.02.2025).

3. Комп'ютерні ігри та мультимедіа як інноваційний підхід до комунікації - 2024 / Матеріали IV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів і студентів, Одеса, 26-27 вересня 2024 р. - Одеса, Видавництво ОНТУ, 2024 р. – 400 с.

4. Speedrun.com – сайт спідрану [Електронний ресурс]. – Режим доступу: https://www.speedrun.com/support/learn/about-speedrun (дата звернення: 25.02.2025).

5. LiveSplit – офіційний сайт [Електронний ресурс]. – Режим доступу: https://livesplit.org/ (дата звернення: 25.02.2025).

6. WSplit – таймер для спідрану [Електронний ресурс]. – Режим доступу: https://sourceforge.net/projects/wsplit/ (дата звернення: 2.02.2025).

7. Splitty – легкий таймер для спідрану [Електронний ресурс]. – Режим доступу: https://github.com/morbOS/splitty (дата звернення: 5.02.2025).

8. SpeedrunIGT – таймер внутрішнього часу гри [Електронний ресурс]. – Режим доступу: https://github.com/redlimerl/SpeedrunIGT (дата звернення: 10.02.2025).

9. OpenGL Development for Minecraft Modding / R. Johnson. – Apress, 2021. – 320 p.

10. Game Configuration and Scripting with JSON and YAML / M. White. –

O'Reilly Media, 2020. – 256 p.

11. Real-Time Event Handling in Games / C. Dawson. – CRC Press, 2019. – 400 p.

12. Minecraft Mapping Tools – інструменти [Електронний ресурс]. – Режим доступу: https://wiki.fabricmc.net/ru:tutorial:mappings (дата звернення 15.04.2025).

13. Database Systems: A Practical Approach to Design, Implementation, and Management / T. Connolly, C. Begg. – Pearson, 2015. – 1440 p.

14. High-Performance Java Code Optimization / J. Bloch. – Addison-Wesley, 2020. – 368 p.

15. Networking and Online Games: Understanding and Engineering Multiplayer Internet Games / D. Griffiths, J. Cowling. – Wiley, 2010. – 400 p.

16. Computer Security: Principles and Practice / W. Stallings, L. Brown. – Pearson, 2017. – 800 p.

17. UML Distilled: A Brief Guide to the Standard Object Modeling Language / M. Fowler. – Addison-Wesley, 2003. – 208 p.

18. https://www.drawio.com/

19. SpongePowered Mixin [Електронний ресурс]. – Режим доступу: https://github.com/SpongePowered/Mixin (дата звернення: 21.03.2025).

20. Minecraft Tick System Overview [Електронний ресурс]. – Режим доступу: https://minecraft.fandom.com/wiki/Tick (дата звернення: 24.03.2025).

21. Cloth Config API [Електронний ресурс]. – Режим доступу: https://github.com/shedaniel/cloth-config (дата звернення: 24.03.2025).

22. Fabric Event API [Електронний ресурс]. – Режим доступу: https://fabricmc.net/wiki/tutorial:event (дата звернення: 27.03.2025).

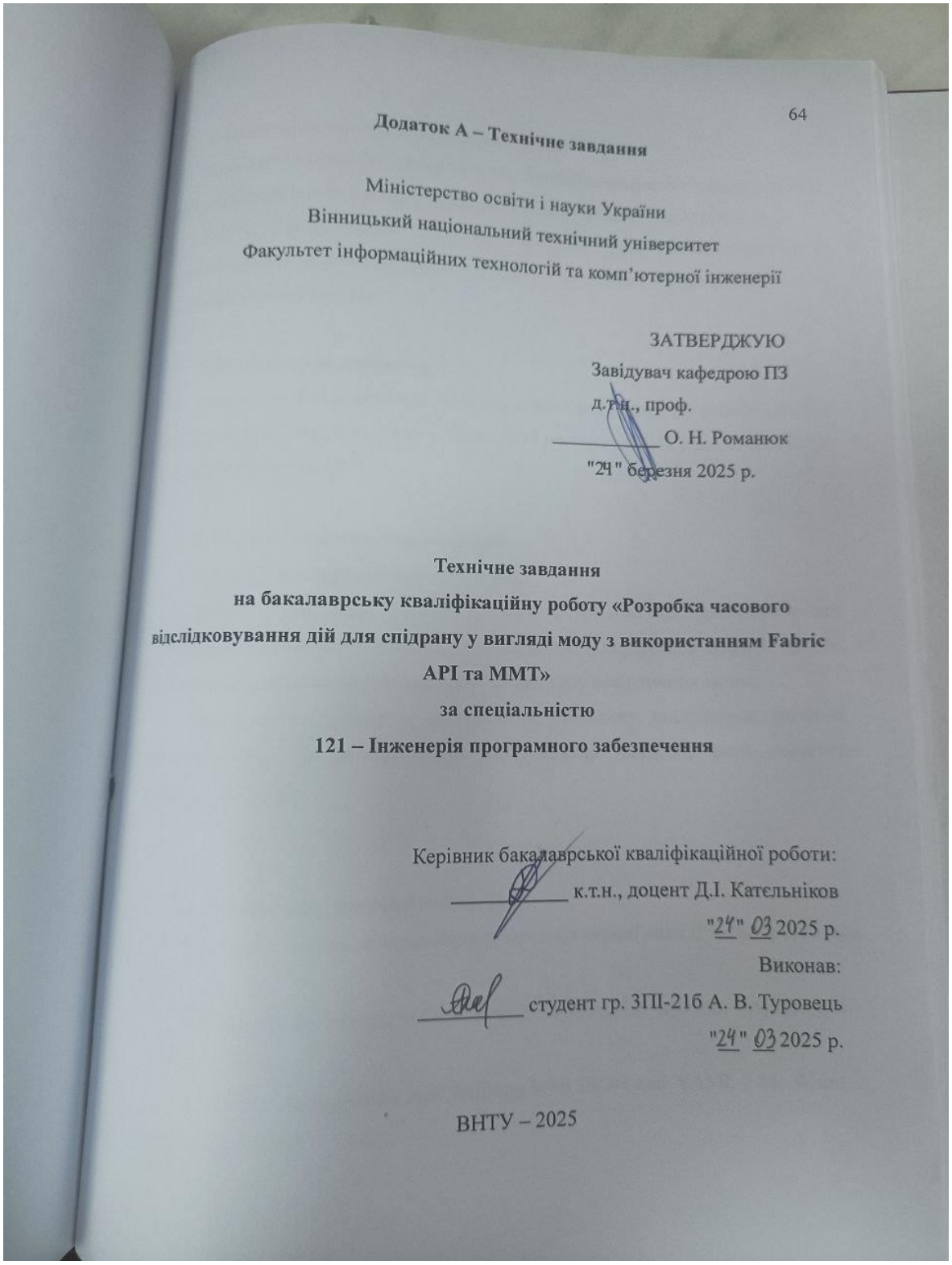
23. IntelliJ IDEA [Електронний ресурс]. – Режим доступу: https://www.jetbrains.com/idea/ (дата звернення: 27.03.2025).

24. MiniHUD Mod [Електронний ресурс]. – Режим доступу: https://www.curseforge.com/minecraft/mc-mods/minihud (дата звернення: 28.03.2025).

25. Sodium Mod [Електронний ресурс]. – Режим доступу: https://www.curseforge.com/minecraft/mc-mods/sodium (дата звернення: 28.03.2025).

26. Lithium Mod [Електронний ресурс]. – Режим доступу: https://www.curseforge.com/minecraft/mc-mods/lithium (дата звернення: 28.03.2025).

ДОДАТКИ



Додаток А – Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедрою ПЗ
д.т.н., проф.
_____ О. Н. Романюк
"24" березня 2025 р.

Технічне завдання
на бакалаврську кваліфікаційну роботу «Розробка часового відслідковування дій для спідрану у вигляді моду з використанням Fabric API та MMT»
за спеціальністю
121 – Інженерія програмного забезпечення

Керівник бакалаврської кваліфікаційної роботи:
_____ к.т.н., доцент Д.І. Катєльніков
"24" 03 2025 р.
Виконав:
_____ студент гр. ЗПІ-21б А. В. Туровець
"24" 03 2025 р.

ВНТУ – 2025

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: «Розробка часового відслідковування дій для спідрану у вигляді моду з використанням Fabric API та MMT».

Галузь застосування – комп'ютерні ігри, геймінг-аналітика, автоматизоване відслідковування ігрових процесів, інструментарій для спідранерської спільноти.

2. Підстава для розробки.

Підставою для виконання бакалаврської кваліфікаційної роботи (БКР) є індивідуальне завдання на БКР та наказ №97 ректора по ВНТУ від 20.03.2025 р. про закріплення тем БКР.

3. Мета та призначення розробки.

Метою бакалаврської кваліфікаційної

роботи є створення моду для точного та гнучкого відслідковування часу проходження Minecraft у спідрані, з інтеграцією у внутрішній ігровий процес, підтримкою різних категорій спідрану та експорту аналітичних даних.

Призначення роботи – забезпечити спільноту спідранерів зручним, кастомізованим та високоточним інструментом таймера, який повністю інтегрується у гру.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БДР:

1. OpenGL Development for Minecraft Modding / R. Johnson. – Apress, 2021. – 320 p.

2. Game Configuration and Scripting with JSON and YAML / M. White. – O'Reilly Media, 2020. – 256 p.

3. Real-Time Event Handling in Games / C. Dawson. – CRC Press, 2019. – 400 p.

4. Networking and Online Games: Understanding and Engineering Multiplayer Internet Games / D. Griffiths, J. Cowling. – Wiley, 2010. – 400 p.

5. Технічні вимоги

Вихідні дані до роботи: відлік часу у тиках Minecraft; облік ключових ігрових подій (перехід у виміри, перемога, смерть тощо); графічний інтерфейс на основі Fabric API; підтримка налаштувань (категорії спідрану, зовнішній вигляд таймера); збереження даних у форматах JSON, CSV; інтеграція з Minecraft через Event Bus Fabric та MMT; сумісність з модами MiniHUD, Lithium, Sodium; середовище розробки – Java, Fabric API, IntelliJ IDEA.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

1. Пояснювальна записка до БДР;
2. Технічне завдання;
3. Лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

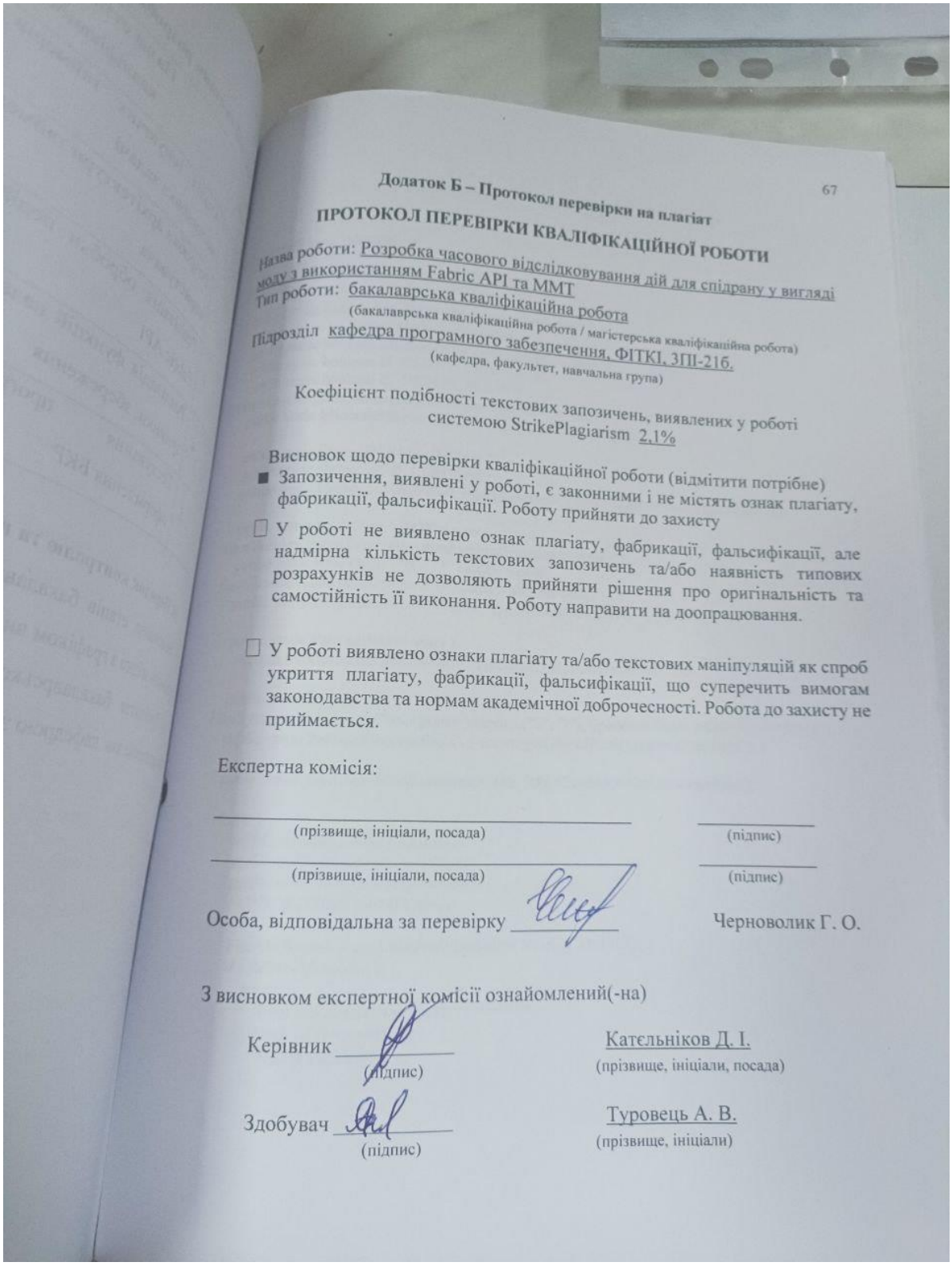
9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз існуючих таймерів для ігор та постановка задачі	25.03.25 – 08.04.25
2	Розробка архітектури таймера та інтерфейсу користувача	09.04.25 – 30.04.25
3	Реалізація обробки подій Minecraft через Fabric API	31.04.25 –18.05.25
4	Реалізація функцій налаштування, запуску, зупинки, збереження	19.05.25 –22.05.25
5	Тестування програмного модуля, оформлення БКР	23.05.25– 30.05.25

10. Порядок контролю та прийняття

Виконання етапів бакалаврської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.



Додаток Б – Протокол перевірки на плагіат

67

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка часового відслідковування дій для спідрану у вигляді моду з використанням Fabric API та MMT

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра програмного забезпечення, ФІТКІ, ЗПІ-21б.
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 2,1%

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

■ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____ _____
(прізвище, ініціали, посада) (підпис)

_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Черноволик Г. О.

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Катєльніков Д. І.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Туровець А. В.
(підпис) (прізвище, ініціали)

Додаток В – Лістинг програмного коду

```
//головний клас таймера

public class SpeedrunTimer implements ModInitializer {

    public static final String MOD_ID = "speedruntimer";
    private static boolean isInitialized = false;
    public static boolean isInitialized() { return isInitialized; }
    public static boolean IS_CLIENT_SIDE = false;
    public static boolean IS_DEBUG_MODE =
FabricLoader.getInstance().isDevelopmentEnvironment();
    public static MinecraftServer DEDICATED_SERVER = null;

    public static String DEBUG_DATA = "";
    public static String MOD_VERSION;
    public static HashMap<Identifier, FontIdentifier> FONT_MAPS = new HashMap<>();

    public static final Gson GSON = new
GsonBuilder().serializeNulls().disableHtmlEscaping().create();
    public static final Gson PRETTY_GSON = new
GsonBuilder().setPrettyPrinting().serializeNulls().create();
    public static final Path FONT_PATH = getGlobalPath().resolve("fonts");

    public static Path getMainPath() {
        return FabricLoader.getInstance().getGameDir().resolve(MOD_ID);
    }
    public static Path getGlobalPath() { return new
File(System.getProperty("user.home").replace("\\", "/"), SpeedrunTimer.MOD_ID).toPath(); }
    public static Path getRecordsPath() { return getGlobalPath().resolve("records"); }

    public static final Set<ModContainer> API_PROVIDERS = Sets.newHashSet();

    static {
        getMainPath().toFile().mkdirs();
        getGlobalPath().toFile().mkdirs();
        getRecordsPath().toFile().mkdirs();
        FONT_PATH.toFile().mkdirs();

        File oldWorlds = getMainPath().resolve("worlds").toFile();
        if (oldWorlds.exists()) {
            try {
                FileUtils.deleteDirectory(oldWorlds);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

```
public boolean isCompleted(MinecraftServer server) {
   Set<String> placedBlocks = Sets.newHashSet();

   for (Item item : getAllItems()) {
      for (ServerPlayerEntity player : server.getPlayerManager().getPlayerList()) {
         if (player.getStatHandler().getStat(Stats.USED, item) > 0) {
            placedBlocks.add(Registry.ITEM.getId(item).toString());
            break;
         }
      }
   }

   return placedBlocks.size() >= 650;
}

private static boolean isIncludedAllBlocks(Block block) {
   if (block.getLootTableId() == LootTables.EMPTY) return false;
   if (block == Blocks.NETHER_PORTAL) return false;
   if (block == Blocks.FARMLAND) return false;
   if (block == Blocks.GRASS_PATH) return false;
   if (block == Blocks.CHORUS_PLANT) return false;
   if (block == Blocks.LILY_PAD) return false;
   if (block == Blocks.PISTON_HEAD) return false;
   if (block == Blocks.TALL_SEAGRASS) return false;
   if (block == Blocks.NETHER_WART) return false;
   if (block == Blocks.SPAWNER) return false;
   if (block == Blocks.BUBBLE_COLUMN) return false;
   if (block == Blocks.BAMBOO_SAPLING) return false;
   if (block == Blocks.PETRIFIED_OAK_SLAB) return false;
   if (block == Blocks.FROSTED_ICE) return false;
   if (block == Blocks.PLAYER_HEAD) return false;
   if (block instanceof InfestedBlock) return false;

   if (block == Blocks.REDSTONE_WIRE) return false;
   if (block == Blocks.TRIPWIRE) return false;
   if (block == Blocks.SWEET_BERRY_BUSH) return false;
   if (block == Blocks.KELP) return false;
   if (block == Blocks.COCOA) return false;
   if (block instanceof FluidBlock) return false;
   if (block instanceof CropBlock) return false;
   if (block instanceof AttachedStemBlock) return false;
   if (block instanceof StemBlock) return false;

   return block != Blocks.AIR;
}

private static List<Item> getItemsForAllBlocks() {
   ArrayList<Item> items = Lists.newArrayList();

   items.add(Items.SWEET_BERRIES);
   items.add(Items.NETHER_WART);
   items.add(Items.KELP);
```

```
        items.add(Items.REDSTONE);
        items.add(Items.STRING);
        items.add(Items.WATER_BUCKET);
        items.add(Items.LAVA_BUCKET);
        items.add(Items.WHEAT_SEEDS);
        items.add(Items.BEETROOT_SEEDS);
        items.add(Items.POTATO);
        items.add(Items.CARROT);
        items.add(Items.MELON_SEEDS);
        items.add(Items.PUMPKIN_SEEDS);
        items.add(Items.COCOA_BEANS);

        return items;
    }



public class RunCategories {
    public static RunCategory ERROR_CATEGORY = new RunCategory("unknown","mc");

    public static Function<InGameTimer, Boolean> anyPercentRetime = timer ->
            !SpeedRunOption.getOption(SpeedRunOptions.TIMER_LEGACY_IGT_MODE) &&
!timer.isCoop() && timer.getRunType() == RunType.RANDOM_SEED && !timer.isRTAMode()
&&
                    (SpeedRunOption.getOption(SpeedRunOptions.ALWAYS_USE_AUTO_RETIME) ||
timer.getInGameTime(false) < 1000 * 60 * 13);
    public static RunCategory ANY = RunCategoryBuilder.create("ANY", "mc",
"speedruntimer.option.timer_category.any")
            .setRetimeFunction(anyPercentRetime).build();
    public static RunCategory CUSTOM = new RunCategory("CUSTOM","mc#");
    public static RunCategory HIGH = new RunCategory("HIGH","mcce#High");
    public static RunCategory KILL_ALL_BOSSES = new
RunCategory("KILL_ALL_BOSSES","mcce#Kill_Bosses");
    public static RunCategory KILL_WITHER = new
RunCategory("KILL_WITHER","mcce#Kill_Bosses");
    public static RunCategory KILL_ELDER_GUARDIAN = new
RunCategory("KILL_ELDER_GUARDIAN","mcce#Kill_Bosses");
    public static RunCategory ALL_ADVANCEMENTS =
RunCategoryBuilder.create("ALL_ADVANCEMENTS","mc#All_Advancements",
"speedruntimer.option.timer_category.all_advancements")
            .setCanSegment(true).build();
    public static RunCategory HALF = new RunCategory("HALF","mcce#Half");
    public static RunCategory POGLOOT_QUATER = new
RunCategory("POGLOOT_QUATER","pogloot_ce#Quater");
    public static RunCategory HOW_DID_WE_GET_HERE = new
RunCategory("HOW_DID_WE_GET_HERE","mcce#How_Did_We_Get_Here",
"advancements.nether.all_effects.title");
    public static RunCategory HERO_OF_VILLAGE = new
RunCategory("HERO_OF_VILLAGE","mcce#Hero_of_the_Village",
"advancements.adventure.hero_of_the_village.title");
    public static RunCategory ARBALISTIC = new
```

```
RunCategory("ARBALISTIC","mcce#Arbalistic", "advancements.adventure.arbalistic.title");
   public static RunCategory COVER_ME_IN_DEBRIS = new
RunCategory("COVER_ME_IN_DEBRIS","mcce#Cover_Me_in_Debris",
"advancements.nether.netherite_armor.title");
   public static RunCategory ENTER_NETHER = new
RunCategory("ENTER_NETHER","mcce#Enter_Nether");
   public static RunCategory ENTER_END = new
RunCategory("ENTER_END","mcce#Etner_Edn");
   public static RunCategory ALL_SWORDS = new
RunCategory("ALL_SWORDS","mcce#All_Swords");
   public static RunCategory ALL_MINERALS = new
RunCategory("ALL_MINERALS","mcce#All_Minerals");
   public static RunCategory FULL_IA_15_LVL = new
RunCategory("FULL_IA_15_LVL","mcce#Full_Iron_Armor_and_15_Levels");
   public static RunCategory ALL_WORKSTATIONS = new
RunCategory("ALL_WORKSTATIONS","mcce#All_Workstations");
   public static RunCategory FULL_INV = new
RunCategory("FULL_INV","mcce#Full_Inventory");
   public static RunCategory STACK_OF_LIME_WOOL = new
RunCategory("STACK_OF_LIME_WOOL","mcce#Stack_of_Lime_Wool");
   public static RunCategory ALL_PORTALS =
RunCategoryBuilder.create("ALL_PORTALS","mcce#All_Portals",
"speedruntimer.option.timer_category.all_portals")
         .setCanSegment(true).build();
   public static AllBlocksRunCategory ALL_BLOCKS = new AllBlocksRunCategory();
   public static RunCategory MINE_A_CHUNK = new
RunCategory("MINE_A_CHUNK","mcce#Mine_a_Chunk");

   public static void checkAllBossesCompleted() {
      InGameTimer timer = InGameTimer.getInstance();
      if (timer.getCategory() == KILL_ALL_BOSSES) {
         if (timer.getMoreData(0) + timer.getMoreData(1) + timer.getMoreData(2) == 3) {
            InGameTimer.complete();
         }
      }
   }
}



   public static List<Item> getAllItems() {
      ArrayList<Item> items = Lists.newArrayList();

      for (Item item : Registry.ITEM) {
         if (getItemsForAllBlocks().contains(item)) items.add(item);

         if (item instanceof BlockItem) {
            BlockItem blockItem = (BlockItem) item;

            if (isIncludedAllBlocks(blockItem.getBlock())) items.add(item);
         }
      }
```

```
        return items;
    }
}



    @Override
    public void onInitialize() {
        MOD_VERSION =
(FabricLoader.getInstance().getModContainer(SpeedrunTimer.MOD_ID).isPresent()
                ?
FabricLoader.getInstance().getModContainer(SpeedrunTimer.MOD_ID).get().getMetadata().getV
ersion().getFriendlyString() : "Unknown+Unknown");

        // звичайні категорії
        new CategoryRegistryImpl().registerCategories().forEach(RunCategory::registerCategory);
        CategoryCondition.registerCondition(new ConditionsRegistryImpl().registerConditions());


        // регістрація апішки
        for (EntrypointContainer<SpeedrunTimerApi> entryPoint :
FabricLoader.getInstance().getEntrypointContainers("speedruntimer", SpeedrunTimerApi.class))
{
            SpeedrunTimerApi api = entryPoint.getEntrypoint();

            // регістрація соло категорії
            RunCategory singleCategory = api.registerCategory();
            if (singleCategory != null) RunCategory.registerCategory(singleCategory);

            // регістрація онлайн категорій
            Collection<RunCategory> multipleCategories = api.registerCategories();
            if (multipleCategories != null)
multipleCategories.forEach(RunCategory::registerCategory);

            // реєстрація кількох умов
            Map<String, CategoryConditionRegisterHelper> multipleConditions =
api.registerConditions();
            if (multipleConditions != null) CategoryCondition.registerCondition(multipleConditions);

            API_PROVIDERS.add(entryPoint.getProvider());
        }

        // завантаження налаштувань
        SpeedRunOption.init();

        CustomCategoryManager.init();

        // кінець ініціалізації
        isInitialized = true;

        // Set properties
```

```
        System.setProperty("speedruntimer.version", MOD_VERSION.split("\\+")[0]);
        System.setProperty("speedruntimer.record", "");

        // оновлення перевірки
        SpeedrunTimerUpdateChecker.checkUpdate();

        // ініціалізація пакетів
        TimerPackets.init();
    }

    private static final Logger LOGGER = LogManager.getLogger("SpeedrunTimer");
    public static void debug(Object obj) {
        if (IS_DEBUG_MODE) LOGGER.info(obj);
    }
    public static void error(Object obj) { LOGGER.error(obj); }

public class SpeedrunTimerClient implements ClientModInitializer {
    public static TimerDrawer TIMER_DRAWER = new TimerDrawer(true);

    public static KeyBinding timerResetKeyBinding;
    public static KeyBinding timerStopKeyBinding;
    public static boolean isInitialized = false;

    @Override
    public void onInitializeClient() {
        // init default option buttons
        SpeedRunOption.addOptionButtonFactories(new
OptionButtonsImpl().createOptionButtons().toArray(new OptionButtonFactory[0]));

        for (EntrypointContainer<SpeedrunTimerApi> entryPoint :
FabricLoader.getInstance().getEntrypointContainers("speedruntimer", SpeedrunTimerApi.class))
{
            SpeedrunTimerApi api = entryPoint.getEntrypoint();

            OptionButtonFactory singleFactory = api.createOptionButton();
            if (singleFactory != null) SpeedRunOption.addOptionButtonFactories(singleFactory);

            Collection<OptionButtonFactory> multipleFactory = api.createOptionButtons();
            if (multipleFactory != null)
SpeedRunOption.addOptionButtonFactories(multipleFactory.toArray(new
OptionButtonFactory[0]));

            SpeedrunTimer.API_PROVIDERS.add(entryPoint.getProvider());
        }

        isInitialized = true;

        // ініціалізація біндів
        timerResetKeyBinding = KeyBindingRegistry.registerKeyBinding(new KeyBinding(
                "speedruntimer.controls.start_timer",
                InputUtil.Type.KEYSYM,
                GLFW.GLFW_KEY_U,
```

```
            "speedruntimer.title.options"
        ));
        timerStopKeyBinding = KeyBindingRegistry.registerKeyBinding(new KeyBinding(
            "speedrunigt.controls.stop_timer",
            InputUtil.Type.KEYSYM,
            GLFW.GLFW_KEY_I,
            "speedrunigt.title.options"
        ));

        // додавання стилів
        FontUtils.copyDefaultFonts();

        SpeedrunTimer.IS_CLIENT_SIDE = true;

        GameInstance.createInstance();
    }
}

public class FontConfigure {

    public float size;
    public float oversample;
    public float[] shift;
    public String skip;

    public static FontConfigure create() {
        return new FontConfigure(11, 6, new float[] { 0, 0 }, "");
    }

    public static FontConfigure fromJson(String json) {
        JsonObject configure = new JsonParser().parse(json).getAsJsonObject();
        FontConfigure fontConfigure = create();
        if (configure.has("size")) fontConfigure.size = configure.get("size").getAsFloat();
        if (configure.has("oversample")) fontConfigure.oversample =
configure.get("oversample").getAsFloat();
        if (configure.has("shift") && configure.get("shift").isJsonArray()) {
            JsonArray shifts = configure.get("shift").getAsJsonArray();
            if (shifts.size() >= 1) {
                fontConfigure.shift[0] = shifts.get(0).getAsFloat();
            }
            if (shifts.size() >= 2) {
                fontConfigure.shift[1] = shifts.get(1).getAsFloat();
            }
        }
        if (configure.has("size")) fontConfigure.size = configure.get("size").getAsFloat();
        if (configure.has("skip")) fontConfigure.skip = configure.get("skip").getAsString();
        return fontConfigure;
    }

    public FontConfigure(float size, float oversample, float[] shift, String skip) {
        this.size = size;
        this.oversample = oversample;
```

```
        this.shift = shift;
        this.skip = skip;
    }

    @Override
    public String toString() {
        return SpeedrunTimer.PRETTY_GSON.toJson(this);
    }
}

public class FontIdentifier {
    private final File file;
    private final Identifier identifier;
    private final FontConfigure fontConfigure;

    public FontIdentifier(File file, Identifier identifier, FontConfigure fontConfigure) {
        this.file = file;
        this.identifier = identifier;
        this.fontConfigure = fontConfigure;
    }

    public Identifier getIdentifier() {
        return this.identifier;
    }

    public FontConfigure getFontConfigure() {
        return this.fontConfigure;
    }

    public File getFile() {
        return this.file;
    }
}

public class FontUtils {
    public static void addFont(HashMap<Identifier, List<Font>> map, File file, File configFile) {
        FileInputStream fileInputStream = null;
        STBTTFontinfo sTBTTFontinfo = null;
        ByteBuffer byteBuffer = null;
        Throwable throwable = null;

        try {
            fileInputStream = new FileInputStream(file);
            sTBTTFontinfo = STBTTFontinfo.malloc();
            byteBuffer = TextureUtil.readAllToByteBuffer(fileInputStream);
            byteBuffer.flip();
            if (!STBTruetype.stbtt_InitFont(sTBTTFontinfo, byteBuffer)) {
                return;
            }

            Identifier fontIdentifier = new Identifier(SpeedrunTimer.MOD_ID,
file.getName().toLowerCase(Locale.ROOT).replace(".ttf", "").replaceAll(" ", "_").replaceAll("[^a-
```

```
z0-9/._-]", ""));
        ArrayList<Font> fontArrayList = new ArrayList<>();

        FontConfigure fontConfigure;
        if (configFile != null && configFile.exists()) {
            fontConfigure = FontConfigure.fromJson(FileUtils.readFileToString(configFile,
StandardCharsets.UTF_8));
        } else {
            fontConfigure = FontConfigure.create();
        }
        fontArrayList.add(new TrueTypeFont(byteBuffer, sTBTTFontinfo, fontConfigure.size,
fontConfigure.oversample, fontConfigure.shift[0], fontConfigure.shift[1], fontConfigure.skip));
        SpeedrunTimer.FONT_MAPS.put(fontIdentifier, new FontIdentifier(file, fontIdentifier,
fontConfigure));

        fontArrayList.add(new BlankFont());

        map.put(fontIdentifier, fontArrayList);
      } catch (FileNotFoundException e) {
        if (sTBTTFontinfo != null) sTBTTFontinfo.free();
        MemoryUtil.memFree(byteBuffer);
      } catch (IOException throwable1) {
        throwable = throwable1;
      } finally {
        try {
          if (fileInputStream != null) fileInputStream.close();
        } catch (IOException e) {
          if (throwable != null) throwable.addSuppressed(e);
        }
      }
    }

    public static void copyDefaultFonts() {
      copyResourceToFont("calibri_bold.ttf");
      copyResourceToFont("calibri_bold.json");
    }

    private static void copyResourceToFont(String fileName) {
      File fontFile = SpeedrunTimer.FONT_PATH.resolve(fileName).toFile();
      if (!fontFile.exists()) {
        InputStream fontInput = FontUtils.class.getResourceAsStream("/font/"+fileName);
        if (fontInput == null) return;
        try {
          FileOutputStream output = new FileOutputStream(fontFile);

          byte[] buf = new byte[1024];

          int readData;
          while ((readData = fontInput.read(buf)) > 0) {
            output.write(buf, 0, readData);
          }
```

```
                fontInput.close();
                output.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

    public RunCategory(String id, String categoryUrl) {
        this(id, categoryUrl, "speedrunigt.option.timer_category." + id.toLowerCase(Locale.ROOT));
    }

    public RunCategory(String id, String categoryUrl, String translateKey) {
        this(id, categoryUrl, translateKey, null, null);
    }

    public RunCategory(String id, String categoryUrl, String translateKey, @Nullable String
conditionFileName, @Nullable JsonArray conditionJson) {
        this(
                id,
                categoryUrl,
                translateKey,
                conditionFileName,
                conditionJson,
                true,
                false,
                false,
                false,
                (value) -> false
        );
    }

    public RunCategory(
            String id,
            String categoryUrl,
            String translateKey,
            @Nullable String conditionFileName,
            @Nullable JsonArray conditionJson,
            boolean autoStart,
            boolean canSegment,
            boolean customUrl,
            boolean hideCategory,
            Function<InGameTimer, Boolean> retimeFunction
        ) {
        this.id = id;
        this.categoryUrl = categoryUrl;
        this.translateKey = translateKey;
        this.conditionFileName = conditionFileName;
        this.conditionJson = conditionJson;
        this.autoStart = autoStart;
        this.canSegment = canSegment;
```

```
        this.customUrl = customUrl;
        this.hideCategory = hideCategory;
        this.retimeFunction = retimeFunction;
    }

    public String getID() {
        return this.id;
    }

    public String getLeaderboardUrl() {
        return (this.customUrl ? "" : "https://www.speedrun.com/") + this.categoryUrl;
    }

    public boolean canSegment() {
        return this.canSegment;
    }

    public boolean isAutoStart() {
        return this.autoStart;
    }

    public boolean isNeedAutoRetime(InGameTimer timer) {
        return this.retimeFunction.apply(timer);
    }

    public boolean isNeedAutoRetime(InGameTimer timer, Function<InGameTimer, Boolean>
retimeFunction) {
        return retimeFunction.apply(timer);
    }

    public boolean isHideCategory() { return this.hideCategory; }

    public TranslatableText getText() {
        return new TranslatableText(this.translateKey);
    }

    public @Nullable JsonArray getConditionJson() {
        return this.conditionJson;
    }

    public @Nullable String getConditionFileName() {
        return this.conditionFileName;
    }
}


public class RunCategoryBuilder {

    private final String id;
    private final String categoryUrl;
    private final String translateKey;
    private boolean autoStart = true;
```

```
    private boolean canSegment = false;
    private boolean customUrl = false;
    private boolean hideCategory = false;
    private Function<InGameTimer, Boolean> retimeFunction = (value) -> false;

    public static RunCategoryBuilder create(String id, String categoryUrl, String translateKey) {
        return new RunCategoryBuilder(id, categoryUrl, translateKey);
    }

    RunCategoryBuilder(String id, String categoryUrl, String translateKey) {
        this.id = id;
        this.categoryUrl = categoryUrl;
        this.translateKey = translateKey;
    }

    public RunCategory build() {
        return new RunCategory(this.id, this.categoryUrl, this.translateKey,
            null,
            null, this.autoStart, this.canSegment, this.customUrl, this.hideCategory,
this.retimeFunction
        );
    }



    public RunCategoryBuilder setAutoStart(boolean autoStart) {
        this.autoStart = autoStart;
        return this;
    }

    public RunCategoryBuilder setCanSegment(boolean canSegment) {
        this.canSegment = canSegment;
        return this;
    }

    public RunCategoryBuilder setUseCustomUrl(boolean useCustomUrl) {
        this.customUrl = useCustomUrl;
        return this;
    }

    public RunCategoryBuilder setHideCategory(boolean hideCategory) {
        this.hideCategory = hideCategory;
        return this;
    }

    public RunCategoryBuilder setRetimeFunction(Function<InGameTimer, Boolean>
retimeFunction) {
        this.retimeFunction = retimeFunction;
        return this;
    }
}
```

```
public class GameInstance {
    public static final ExecutorService SAVE_MANAGER_THREAD =
Executors.newSingleThreadExecutor();
    private static final Logger LOGGER = LogManager.getLogger("Game Instance");
    private static GameInstance INSTANCE;
    private final List<Event> bufferedEvents;
    private List<Event> events;
    private final Path globalEventsPath;
    private TimerWorld world;

    private GameInstance() {
        this.bufferedEvents = new ArrayList<>();
        this.globalEventsPath = SpeedrunTimer.getGlobalPath().resolve("latest_world.json");
    }

    public static GameInstance getInstance() {
        return INSTANCE;
    }

    public static void createInstance() {
        if (INSTANCE == null) {
            INSTANCE = new GameInstance();
        }
    }

    private static UUID getLocalPlayerID() {
        return MinecraftClient.getInstance().getSession().getProfile().getId();
    }

    public void tryLoadWorld(String worldName) {
        if (!SpeedrunTimer.IS_CLIENT_SIDE) {
            return;
        }
        File worldFile = InGameTimerUtils.getTimerLogDir(worldName, "");
        if (worldFile != null) {
            this.loadWorld(worldFile.toPath());
            LOGGER.info("Loaded events world.");
            checkJoinEvents();
        } else {
            LOGGER.error("Didn't load events world.");
        }
    }

    private void checkJoinEvents() {
        // повторне підключення
        if (this.events.stream().anyMatch(event -> event.type.equals("leave_world"))) {
            this.callEvents("rejoin_world");
        }

        // перевірка мульплеєру
        if (this.events.stream().anyMatch(event -> event.type.equals("multiplayer"))) return;
```

```
        Set<UUID> previousPlayers = this.world.getPreviousPlayers();
        if (previousPlayers.size() > 1 || (previousPlayers.size() == 1 &&
previousPlayers.stream().noneMatch(uuid -> uuid.equals(getLocalPlayerID())))) {
            this.callEvents("multiplayer");
        }
    }

    public void ensureWorld() {
        if (SpeedrunTimer.IS_CLIENT_SIDE && !this.hasWorldLoaded()) {
            InGameTimer timer = InGameTimer.getInstance();
            LOGGER.info("Attempting event world load at " + timer.getWorldName());
            this.tryLoadWorld(timer.getWorldName());
        }
    }

    private void loadWorld(Path worldTimerDir) {
        this.world = new TimerWorld(worldTimerDir, this.globalEventsPath);
        this.events = this.world.getEventRepository().getOldEvents();
        this.addBufferedEvents();
    }

    public void closeTimer() {
        this.bufferedEvents.clear();
        this.events = null;
        this.world = null;
    }

    private void addBufferedEvents() {
        if (this.world != null && !this.bufferedEvents.isEmpty()) {
            List<Event> buffered = new ArrayList<>(this.bufferedEvents);
            int removed = 0;
            for (Event bufferedEvent : buffered) {
                if (this.canTriggerEvent(bufferedEvent)) {
                    if (this.events != null) {
                        this.events.add(bufferedEvent);
                    } else {
                        // Since the list of old events hasn't loaded yet, we can't know for sure if the event
has been triggered or not, so we just add the buffered events later.
                        LOGGER.error("Couldn't add buffered event to events array.");
                        return;
                    }
                    this.sendEventToRepository(bufferedEvent);
                    this.bufferedEvents.remove(bufferedEvent);
                    removed++;
                }
            }
            LOGGER.info("Loaded " + removed + " buffered event" + (removed != 1 ? "s" : "") + ".");
        }
    }

    public void addEvent(Event event) {
        if (this.world != null && this.events != null) {
```

```
        if (this.canTriggerEvent(event)) {
            this.addBufferedEvents();
            this.events.add(event);
            this.sendEventToRepository(event);
        }
    } else {
        this.bufferedEvents.add(event);
    }
}

public void callEvents(String type) {
    this.callEvents(type, null);
}

public void callEvents(String type, Function<EventFactory, Boolean> condition) {
    if (!SpeedrunTimer.IS_CLIENT_SIDE) return;
    for (EventFactory factory : EventFactoryLoader.getEventFactories(type)) {
        if (condition == null || condition.apply(factory)) {
            this.addEvent(factory.create());
        }
    }
}

public boolean canTriggerEvent(Event e) {
    if (this.events == null) {
        return true;
    }
    boolean repeatable = EventFactoryLoader.isEventRepeatable(e);
    for (Event event : this.events) {
        if (event.id.equalsIgnoreCase(e.id)) {
            if (!repeatable) {
                return false;
            }
            if (event.gameTime.equals(e.gameTime) && event.realTime.equals(e.realTime)) {
                return false;
            }
        }
    }
    return true;
}

public boolean hasWorldLoaded() {
    return this.world != null;
}

private boolean shouldUpdateGlobal(Event event) {
    return this.events.size() > 1 || !event.type.equals("leave_world");
}

private void sendEventToRepository(Event event) {
    if (this.shouldUpdateGlobal(event)) {
        this.world.getEventRepository().add(event);
```

```
        } else {
            this.world.getEventRepository().addOnlyToLog(event);
        }
    }
}


public class TimerWorld {
    private static final Gson GSON = new Gson();
    private static final String EVENT_LOG_FILE_NAME = "events.log";
    private final Path worldFolderPath;
    private final List<String> mods;
    private final GameVersion version;
    private final EventRepository eventRepository;

    TimerWorld(Path worldFolderPath, Path globalEventsPath) {
        this.worldFolderPath = worldFolderPath;
        this.mods = FabricLoader.getInstance().getAllMods().stream().map(mod ->
mod.getMetadata().getId()).collect(Collectors.toList());
        this.version = SharedConstants.getGameVersion();
        this.eventRepository = new EventRepository(this,
this.worldFolderPath.resolve(EVENT_LOG_FILE_NAME), globalEventsPath);
    }

    public String getWorldData() {
        return GSON.toJson(this.getWorldDataObject());
    }

    private JsonObject getWorldDataObject() {
        JsonObject object = new JsonObject();
        JsonArray modsArray = new JsonArray();
        this.mods.forEach(modsArray::add);
        object.addProperty("world_path", this.worldFolderPath.getParent().toString().replace("\\",
"/"));
        object.add("mods", modsArray);
        object.addProperty("version", this.version.getName());
        object.addProperty("mod_version", SpeedrunTimer.MOD_VERSION);
        object.addProperty("category", InGameTimer.getInstance().getCategory().getID());
        return object;
    }

    public EventRepository getEventRepository() {
        return this.eventRepository;
    }

    public Set<UUID> getPreviousPlayers() {
        try {
            return
Arrays.stream(Objects.requireNonNull(worldFolderPath.resolveSibling("stats").toFile().list())).ma
p(s -> UUID.fromString(s.split("\\.")[0])).collect(Collectors.toSet());
        } catch (Exception e) {
```

```
return Collections.emptySet();
```

Додаток Г – Ілюстративна частина

ІЛЮСТРАТИВНА ЧАСТИНА

РОЗРОБКА ЧАСОВОГО ВІДСЛІДКОВУВАННЯ ДІЙ ДЛЯ СПІДРАНУ У ВИГЛЯДІ МОДУ З ВИКОРИСТАННЯМ FABRIC API ТА MMT

Вінницький національний технічний університет
Факультет інформаційних технологій та комп`ютерної інженерії
Кафедра програмного забезпечення

БАКАЛАВРСЬКА ДИПЛОМНА РОБОТА
на тему:

РОЗРОБКА ЧАСОВОГО ВІДСЛІДКОВУВАННЯ ДІЙ ДЛЯ СПІДРАНУ У ВИГЛЯДІ МОДУ З ВИКОРИСТАННЯМ FABRIC API ТА MMT

Виконав:
ст. групи ЗПІ-21б.
Туровець А. В.

Науковий керівник:
к.т.н., доц. каф ПЗ
Катєльніков Д. І.

Рисунок Г.1 – Тема роботи

РОЗРОБКА ЧАСОВОГО ВІДСЛІДКОВУВАННЯ ДІЙ ДЛЯ СПІДРАНУ У ВИГЛЯДІ МОДУ З ВИКОРИСТАННЯМ FABRIC API ТА MMT

Мета роботи: підвищення точності часових вимірів процесу спідрану у грі Minecraft за рахунок розробки інтегрованого таймера, що працюватиме у вигляді моду з використанням Fabric API.

Об’єкт дослідження: процес отримання часових вимірів в процесі спідрану у грі Minecraft.

Предмет дослідження: методи та алгоритми розробки інтегрованого таймера для спідрану для гри Minecraft у вигляді спеціально створеної для змагання арени на основі Fabric API.

Рисунок Г.2 – Мета, об'єкт та предмет дослідження

Про термін "Ігровий світ" (Game World)

Вікіпедія визначає Game world як: «Ігровий світ — це вигадана обстановка, в якій відбувається відеогра. Це загальне середовище, що включає його географію, історію, персонажів та правила, в рамках яких розгортається сюжет гри та діють її механізми.»

Використання у наукових працях

Відомі наукові роботи, як-от Salen, K., & Zimmerman, E. (2004). Rules of Play: Game Design Fundamentals, постійно використовують термін "Game World". Це підтверджує його академічну прийнятність.

Чому "Ігровий світ" краще?

Інші переклади, наприклад "Ігрова віртуальна реальність" (Скородєлов та ін., 2020), є менш лаконічними та поширеними. "Ігровий світ" (Game World) — це точний, зрозумілий і визнаний термін у світовій науковій літературі.

Рисунок Г.3 – Про термін «Ігровий світ»

Наукова новизна отриманих результатів

1. Подальшого розвитку отримав метод побудови таймеру спідрану, який, на відміну від існуючих методів побудови, створює таймер як частину ігрового світу, що дозволяє синхронізувати його з ігровими процесами і, таким чином, досягати більшої точності і уникати впливу зовнішніх факторів.

2. Подальшого розвитку отримав метод інтеграції таймерів у ігровий світ, який на відміну від існуючих методів, використовує розроблену систему модульної інтеграції, що дозволяє легко адаптувати таймер під різні категорії спідрану.

Рисунок Г.4 – Наукова новизна

Практична цінність отриманих результатів

Практична цінність роботи полягає в тому, що на основі отриманих в бакалаврській кваліфікаційній роботі теоретичних положень запропоновано алгоритми та розроблено програмний засіб - таймер для спідранерів Minecraft.

Рисунок Г.6 – Практична цінність

Аналоги

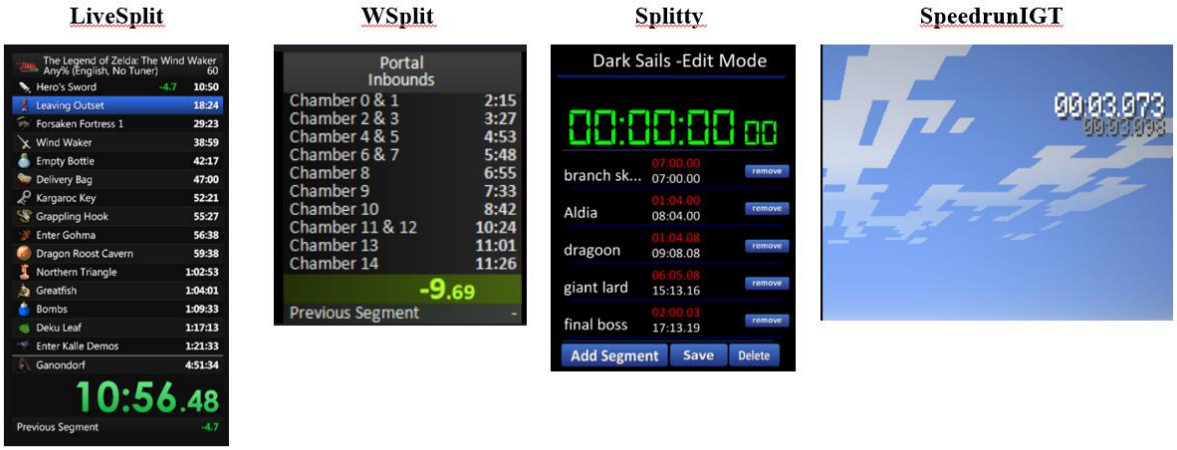


Рисунок Г.7 – Аналоги

Порівняльний аналіз аналогів

Критерій	Live Split	WSplit	Splitty	Speedrun IGT	Власний продукт
Точність часу	1	0.5	0.5	0.5	1
Автоматичні тригери	0	0	0	0	1
Кросплатформенність	0	0	1.5	0	0
Кастомізація	1	0.5	1	1	1.5
Інтеграція з онлайн-сервісами	1	0	0	0	1
Результат	**60%**	**20%**	**60%**	**30%**	**90%**

Рисунок Г.8 – Порівняльний аналіз аналогів

Рисунок Г.9 – Діаграма діяльності

Рисунок Г.10 – Алгоритм функціонування системи

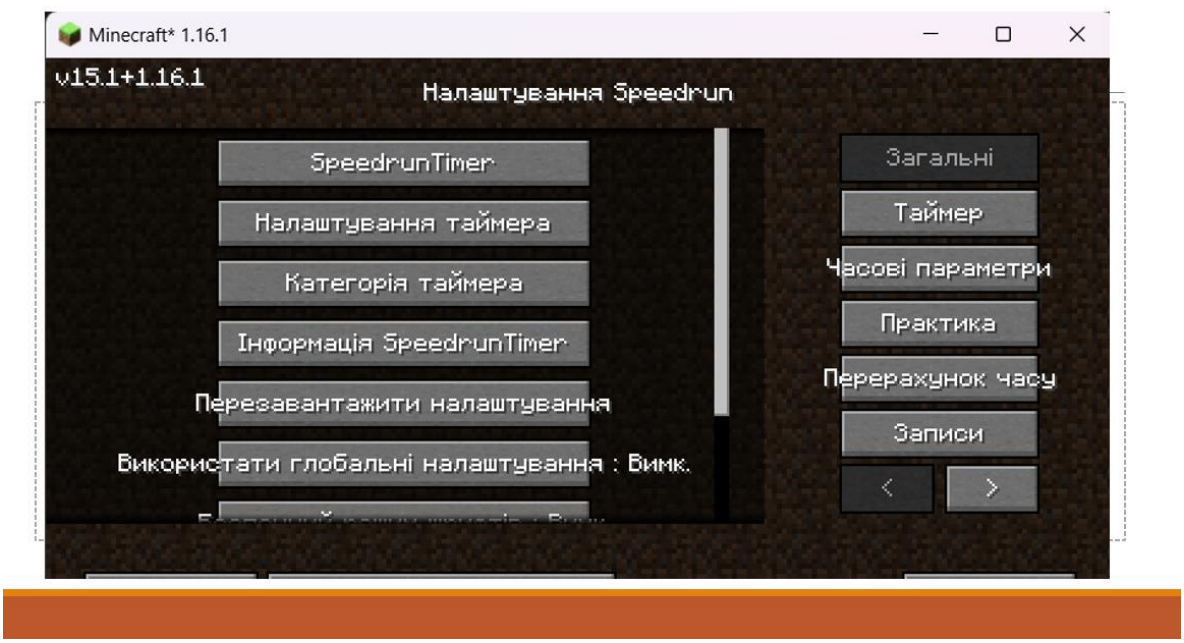


Рисунок Г.11 – Меню керування таймера

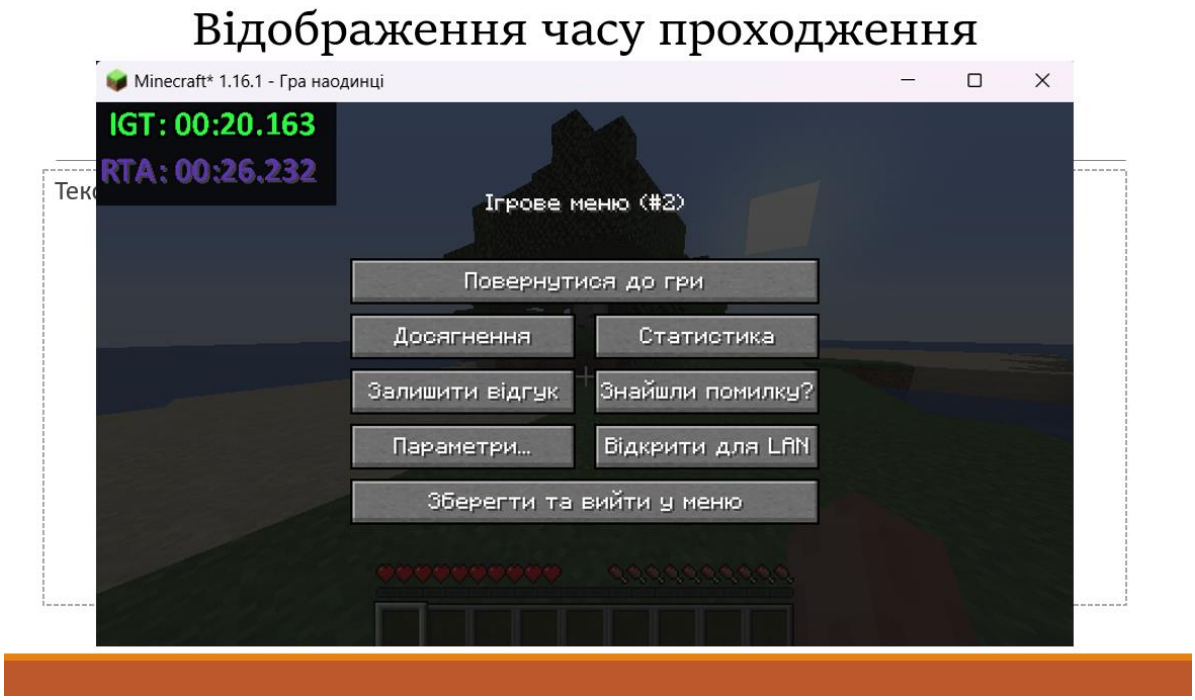


Рисунок Г.12 – Відображення часу проходження

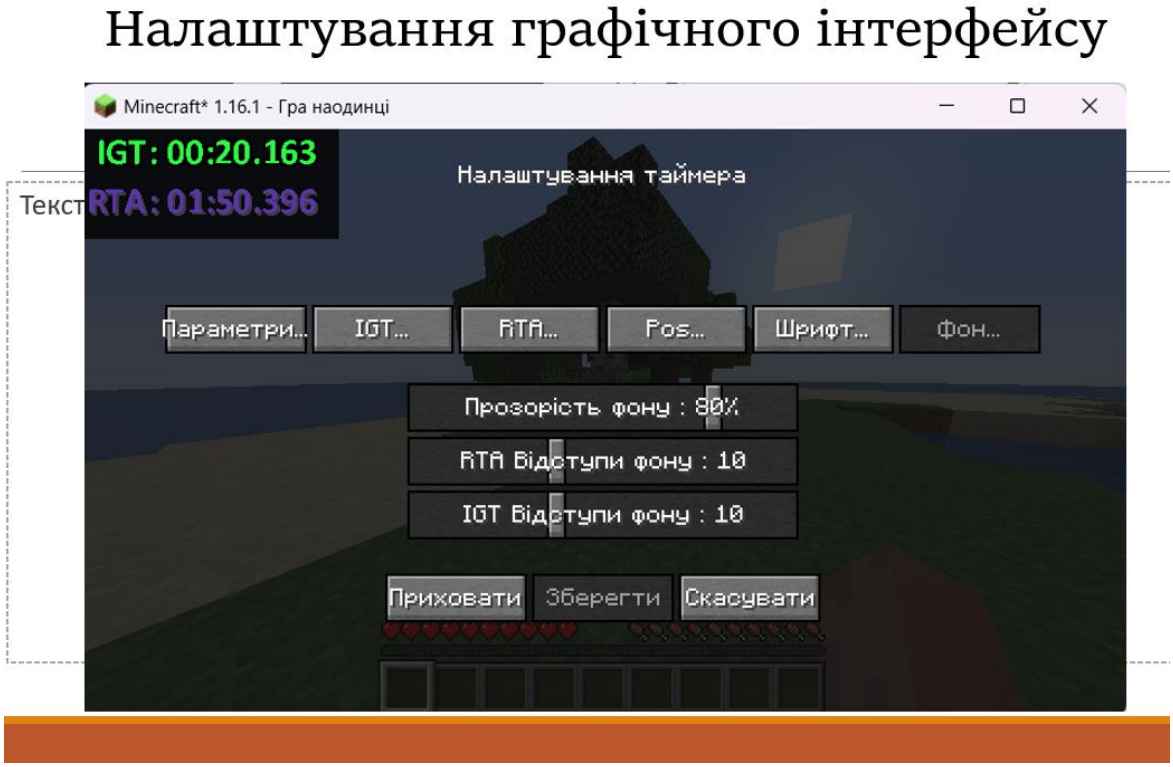


Рисунок Г.13 – Налаштування графічного інтерфейсу

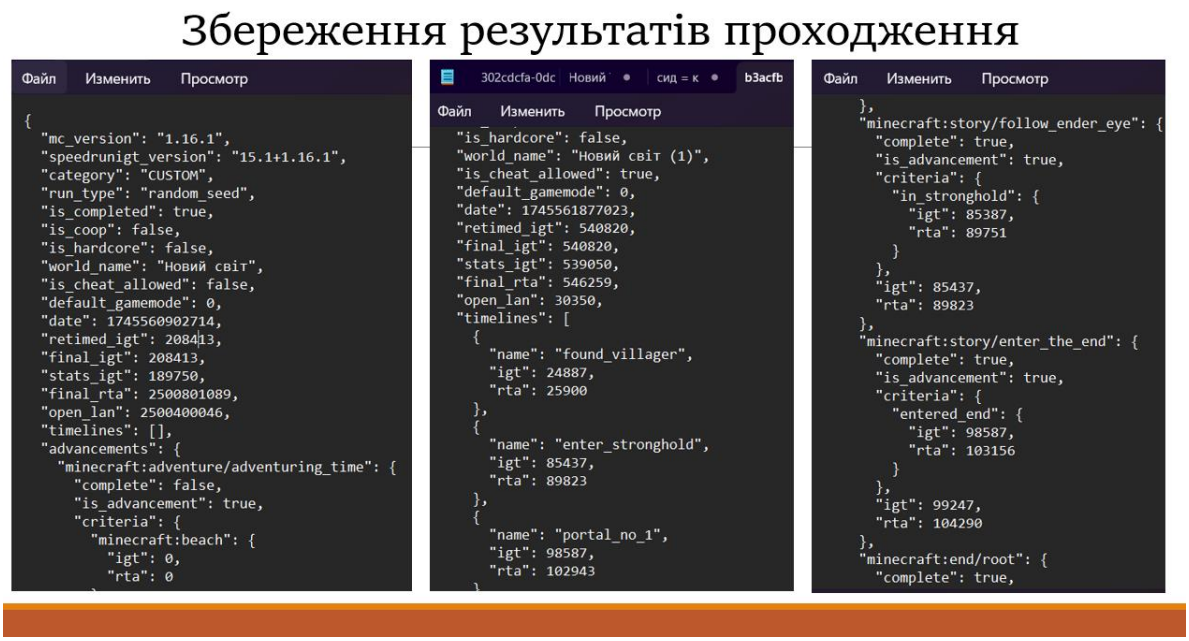


Рисунок Г.14 – Збереження результатів проходження

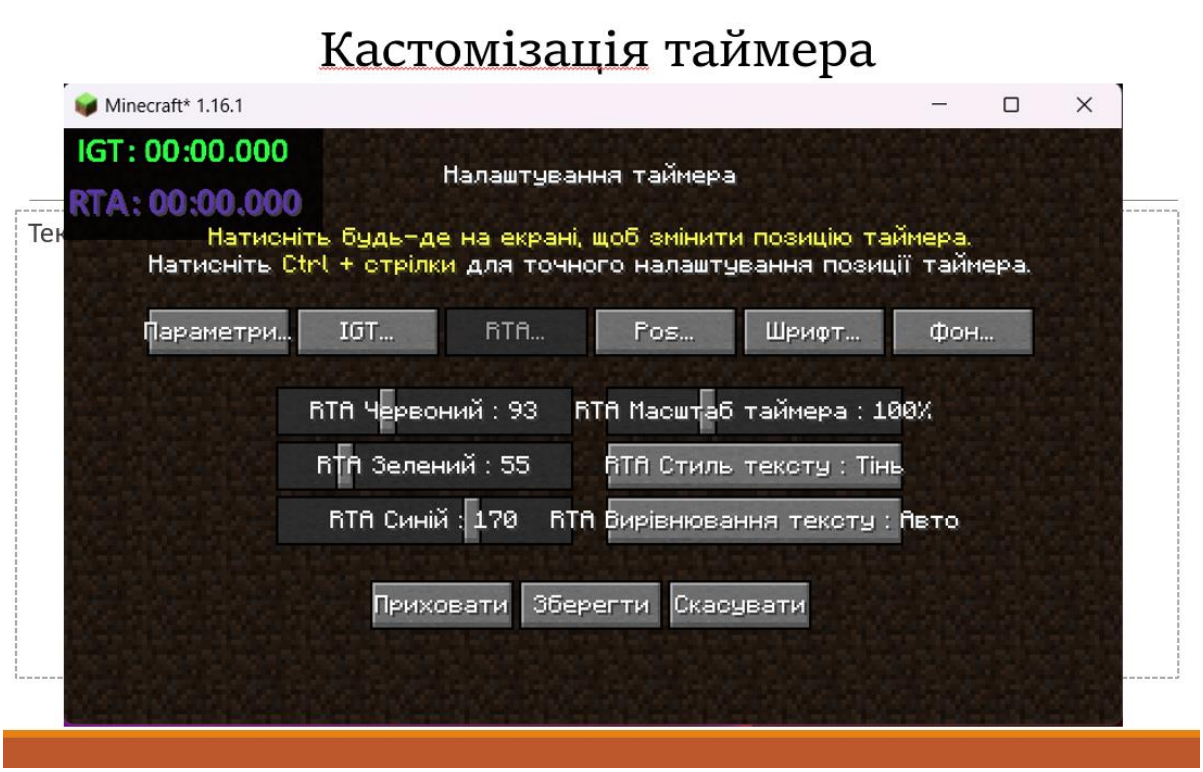


Рисунок Г.15 – Кастомізація таймера

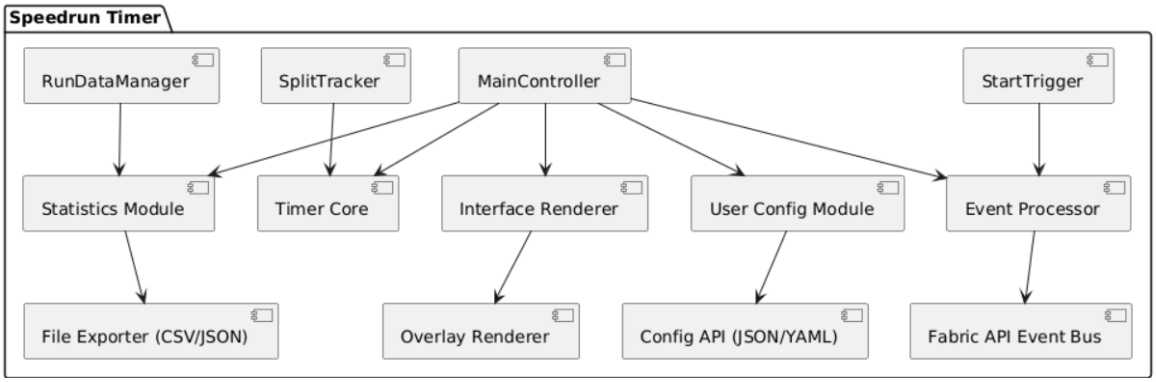


Рисунок Г.16 – Діаграма компонентів системи модульної інтеграції

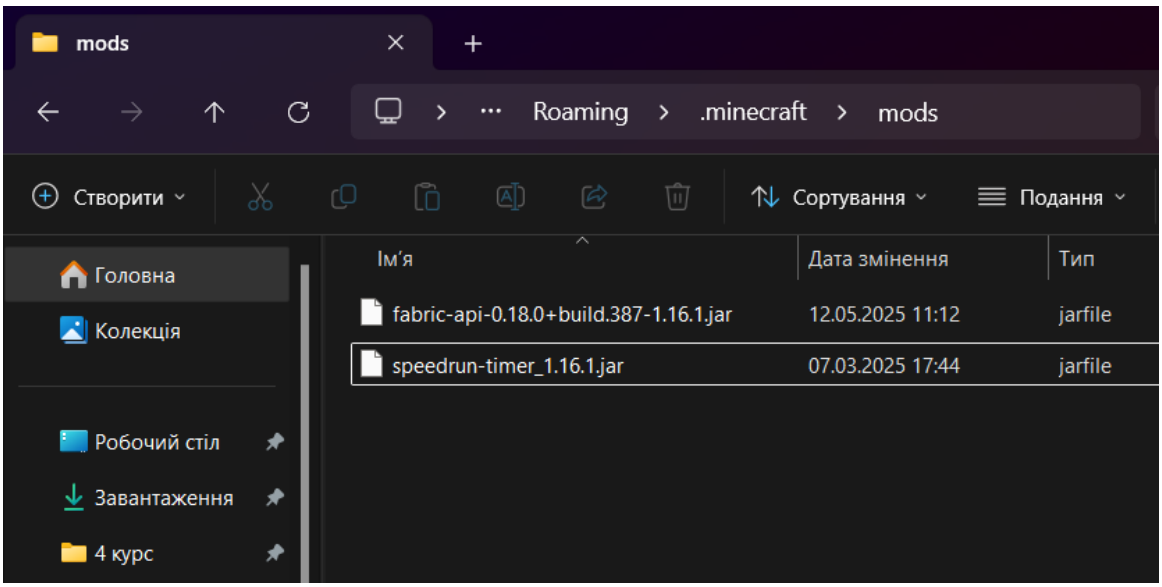


Рисунок Г.17 – Розположення файлу моду

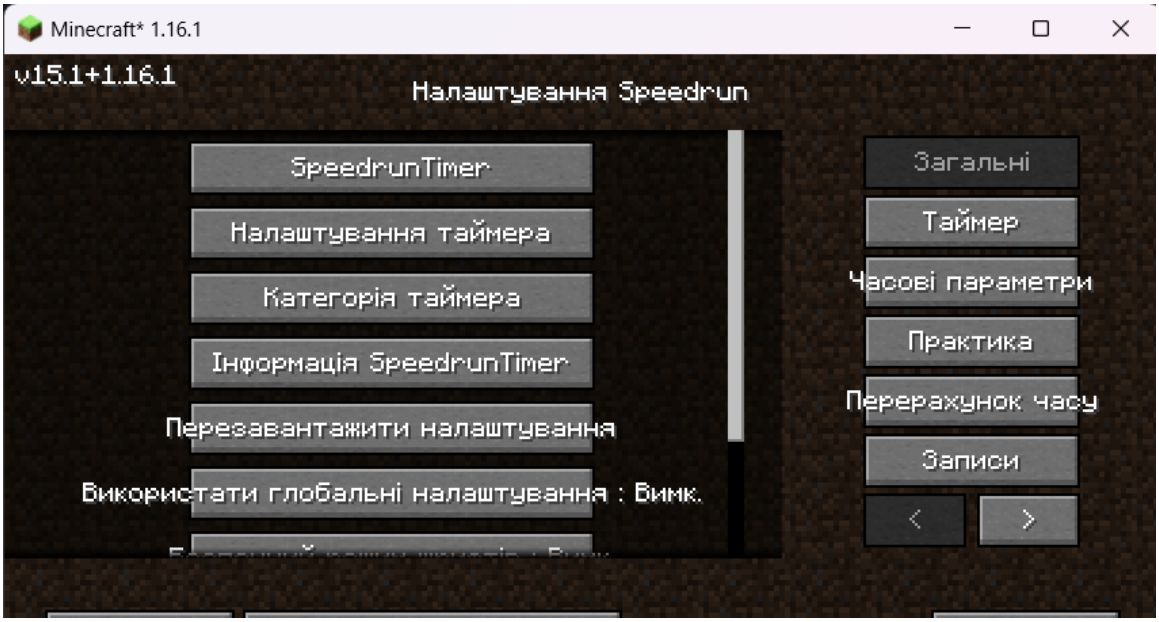


Рисунок Г.18 – Меню керування таймера

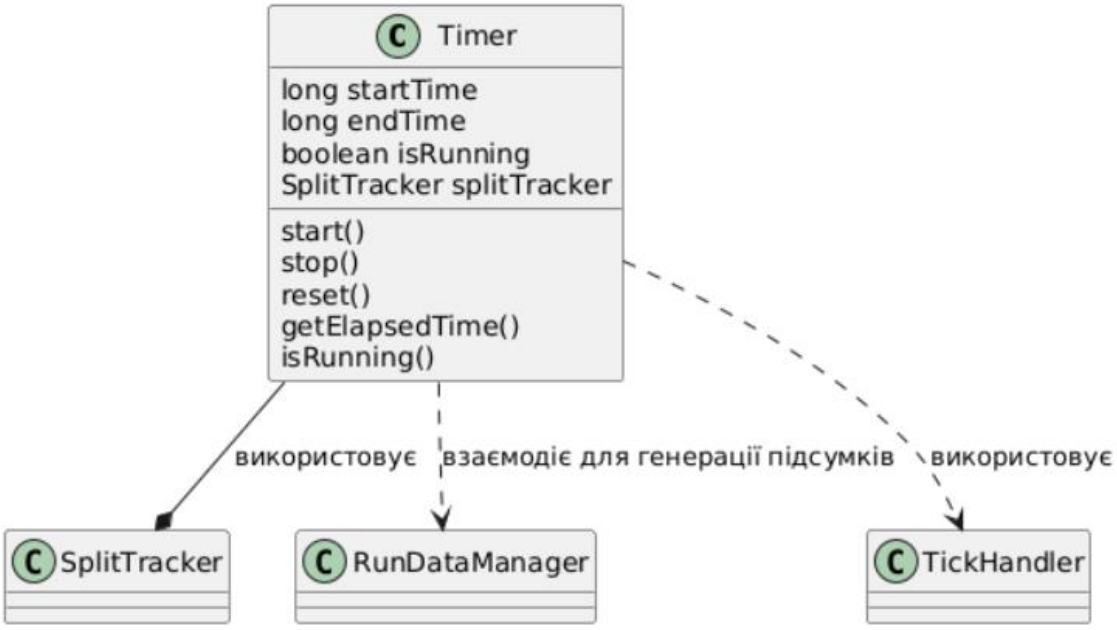


Рисунок Г.19 – Діаграма класу Timer

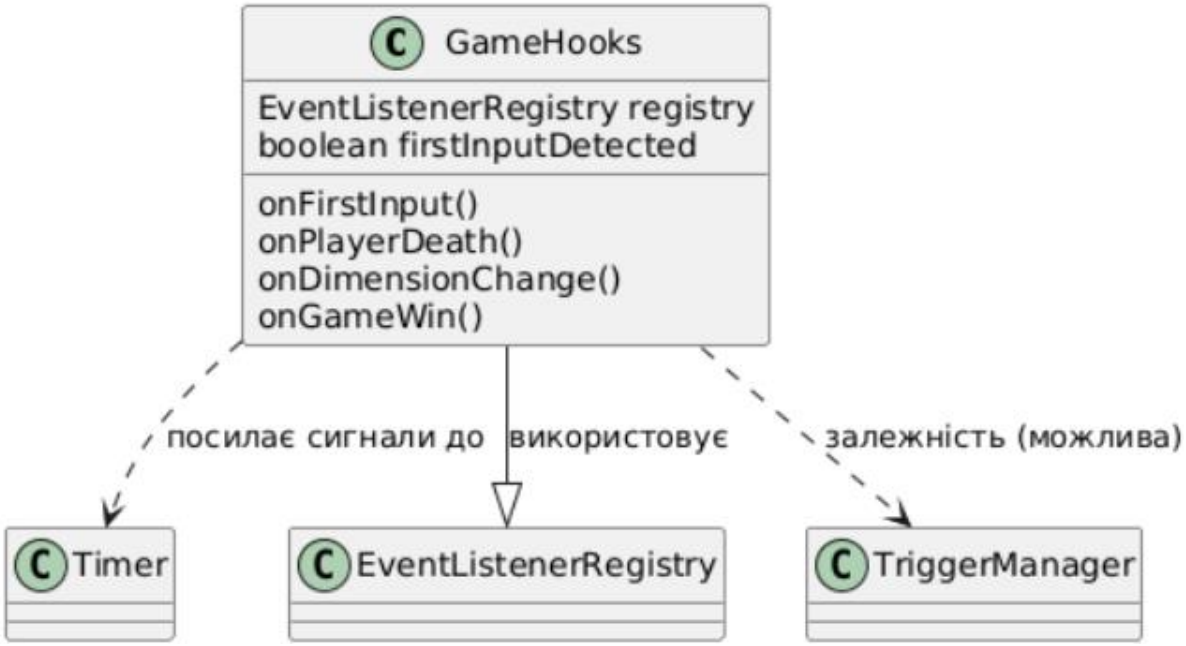


Рисунок Г.20 – Діаграма класу GameHooks

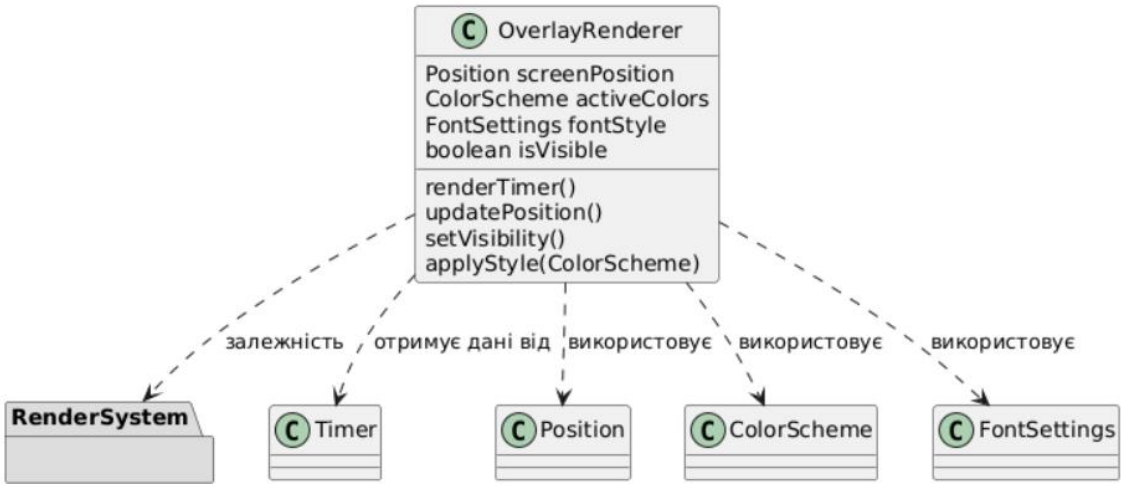


Рисунок Г.21 – Діаграма класу OverlayRenderer

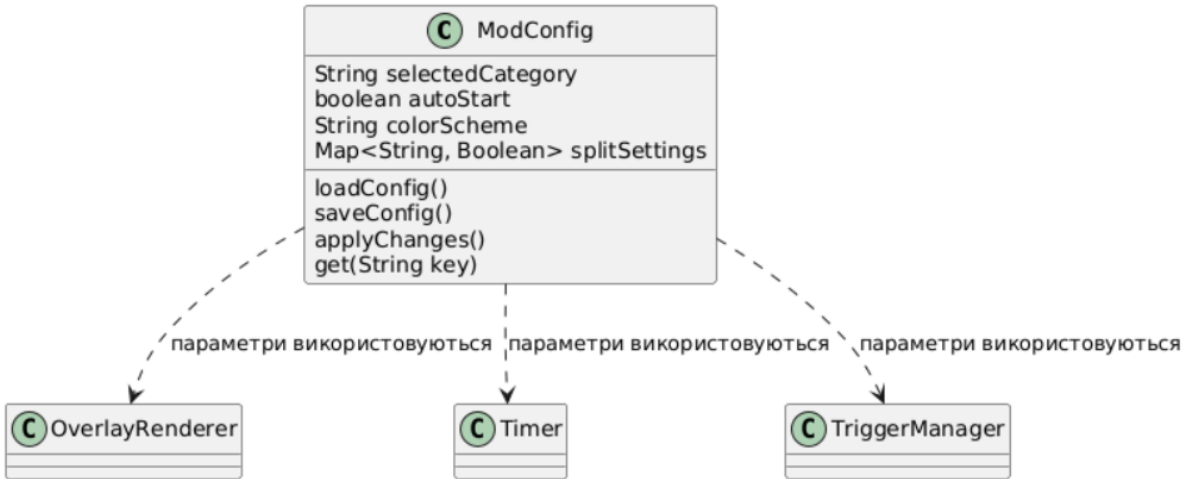


Рисунок Г.22 – Діаграма класу ModConfig

Висновки

- проаналізовано сферу спідрану, ідентифіковано проблеми з точним вимірюванням часу;
- досліджено наявні таймери, виокремлено їхні переваги, обмеження та особливості;
- розроблено власну архітектурну модель на основі модульного підходу, незалежну від зовнішніх засобів фіксації часу;
- обґрунтовано вибір інструментів і технологій: fabric api, java, minecraft mapping tools (mmt);
- реалізовано ключові програмні модулі: ядро обліку часу (timer), обробка ігрових подій (gamehooks), графічний рендеринг інтерфейсу (overlayrenderer), конфігураційний модуль (modconfig) і блок збору статистики (rundatamanager);
- забезпечено підтримку ручного та автоматичного запуску, кастомізацію зовнішнього вигляду, зберігання результатів у форматах csv/json і відновлення даних;
- проведено тестування на коректність, стабільність, продуктивність та сумісність із іншими модифікаціями.

Рисунок Г.23 – Висновки