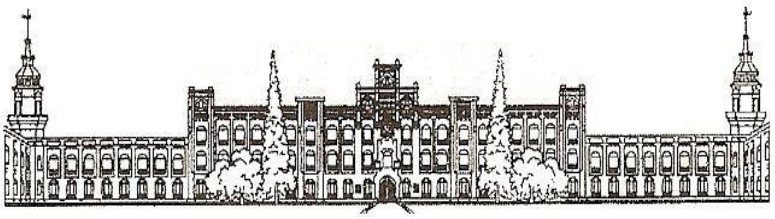

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут» ім. Ігоря Сікорського

Звіт

Лабораторна робота №4

з дисципліни «Технології та інструменти розробки ПЗ»

«Розробка вихідного коду ПЗ»

Виконав студент групи: КВ-32

Косарук Захар

Перевірив: Омельченко Андрій

Київ 2025

Мета роботи: отримати навички застосування інструментів створення та зберігання вихідного коду програмного забезпечення.

Завдання:

1. Із використанням GitHub Project створити Kanban дошку і створити в ній картки із описом завдань, необхідних для створення програмного забезпечення.
2. Із використанням GitHub створити репозиторій для збереження вихідного коду проєкту.
3. Обрати засіб розробки вихідного коду програмного забезпечення.
4. Створити вихідний код програмного забезпечення із використанням системи
контроля версій Git та інструментів Code Review (Pull Requests). При створенні вихідного коду програмного забезпечення відмічати хід роботи із використанням карток Kanban дошки проєкту.
5. Оформити звіт із лабораторної роботи. Зміст звіту:

- титульний аркуш із відомостями про виконавця;
- завдання;
- виконане завдання згідно варіанту (вихідний код, скріншоти Kanban дошки, історія комітів Git);
- висновки.

Хід роботи

1.

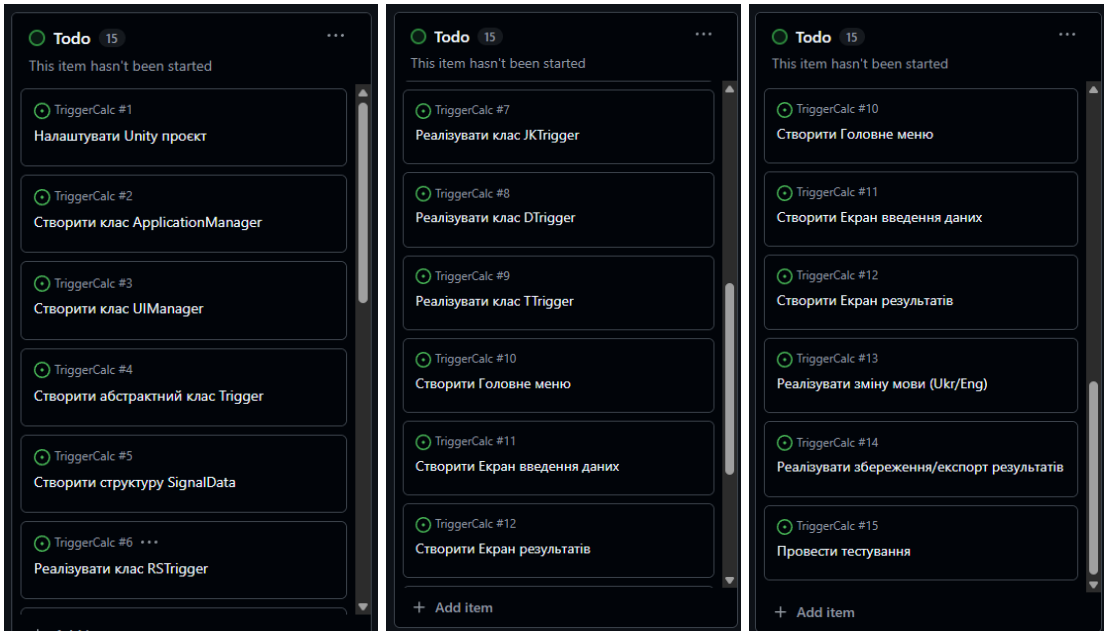

Рисунок 1 – Kanban дошка до початку роботи.

На початковому етапі виконання лабораторної роботи №4 було налаштовано інструменти для управління проєктом та відстеження життєвого циклу розробки.

Відповідно до завдання, було використано інструмент **GitHub Projects** для створення **Kanban-дошки** під назвою «Розробка TriggerCalc». Ця дошка слугуватиме централізованим місцем для візуалізації та менеджменту всіх завдань, пов'язаних із проєктом.

Процес створення дошки включав декомпозицію загального проєкту на менші, керовані завдання (картки). Джерелом для цих завдань слугували документи, розроблені у попередніх лабораторних роботах:

1. **Функціональні та нефункціональні вимоги** з Технічного завдання (Лабораторна робота №2).
2. **Архітектурні компоненти**, визначені у Діаграмі класів (Лабораторна робота №3).

Кожне завдання (наприклад, «Створити клас ApplicationManager», «Реалізувати логіку JKTrigger», «Створити Головне меню» тощо) було створене як окрема **"Issue"** у репозиторії TriggerCalc. Це дозволить у подальшому чітко пов'язувати гілки (branches), коміти (commits) та запити на злиття (Pull Requests) з конкретними картками на дошці.

Усі створені завдання були розміщені у колонці **"Todo"** (До виконання), сформувавши таким чином початковий **беклог (backlog)** проєкту. Це повністю готує робоче середовище до наступного етапу — безпосередньої розробки вихідного коду.

2.

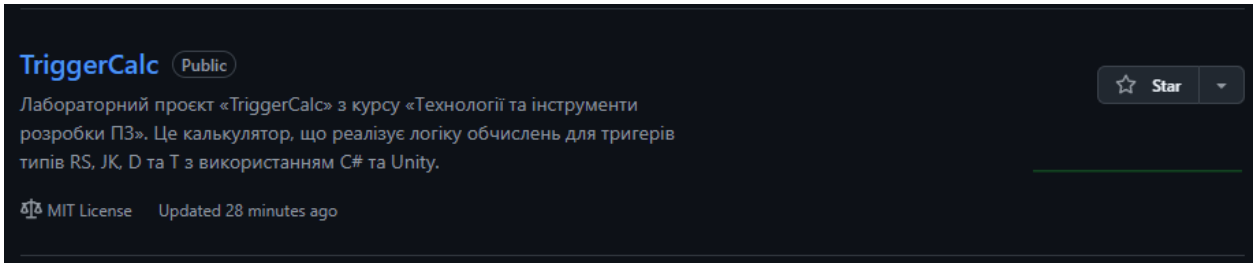

Рисунок 2 – Репозиторій

https://github.com/zZaKko96/TriggerCalc

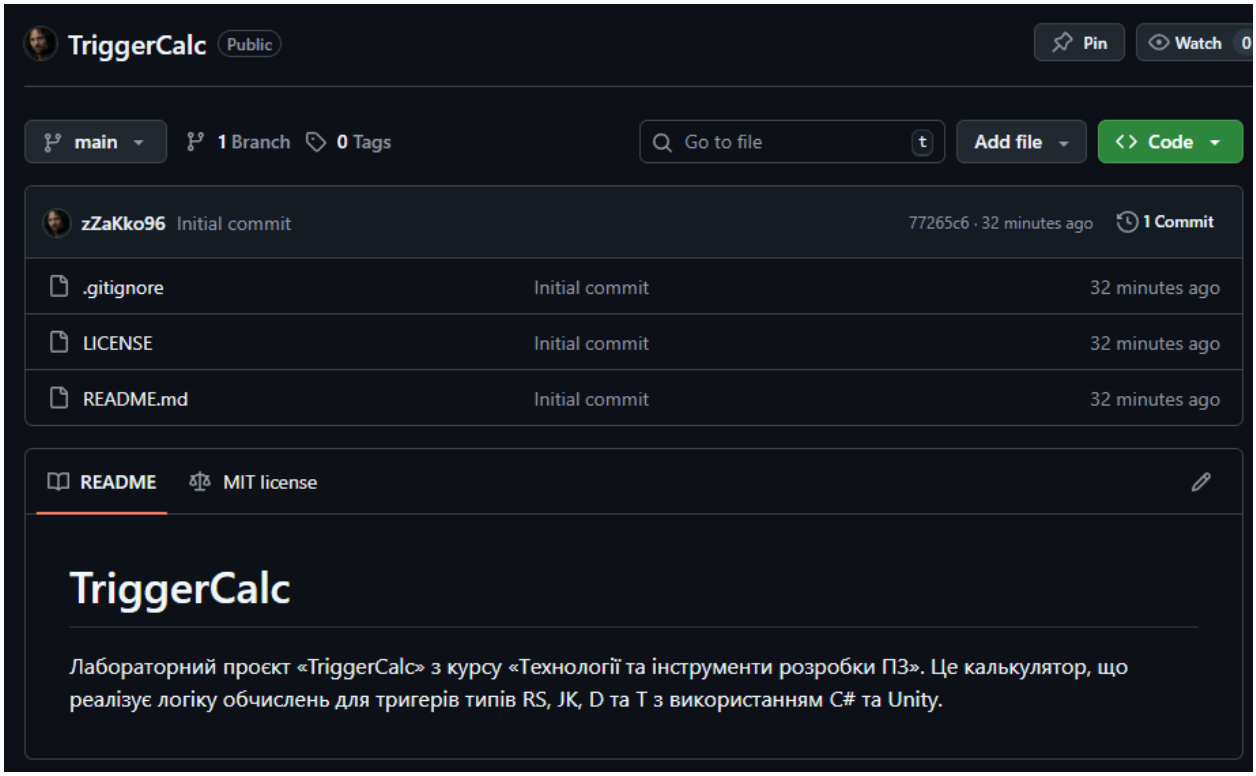

Рисунок 3 - Репозиторій

Для забезпечення централізованого зберігання вихідного коду, відстеження змін та спільної роботи було виконано завдання зі створення репозиторію на платформі **GitHub**.

Репозиторій отримав назву **TriggerCalc**, що відповідає назві проєкту, визначеній у Лабораторній роботі №1.

Під час ініціалізації репозиторій було одразу налаштовано з додаванням трьох ключових файлів:

- **README.md**: Файл з описом проєкту, його освітньої мети та технологічного стеку (C# та Unity), який слугує "обличчям" проєкту.
- **LICENSE**: Додано **MIT License**, яка є дозволяючою (permissive) ліцензією з відкритим вихідним кодом. Вона чітко визначає права та умови використання коду.
- **.gitignore**: Додано стандартний шаблон .gitignore для **Unity**. Цей файл критично важливий, оскільки він автоматично виключає з-під контролю версій тимчасові файли, кеш та специфічні для локальної машини налаштування середовища розробки (наприклад, папки Library/ та Temp/).

Таким чином, завдання №2 виконано, і репозиторій повністю готовий до клонування та прийому перших комітів з вихідним кодом.

3.

Вибір засобів розробки вихідного коду ґрунтувався на вимогах, визначених у Технічному завданні (Лабораторна робота №2), та архітектурі, спроєктованій у Лабораторній роботі №3.

Було обрано наступний технологічний стек:

- **Мова програмування: C#**
- **Середовище та рушій: Unity**

Це рішення обґрунтоване ключовими перевагами, що безпосередньо відповідають поставленим вимогам:

1. **Відповідність архітектурі:** C# є сучасною, потужною об'єктно-орієнтованою мовою. Це ідеально підходить для реалізації архітектури, що була спроєктована в Лабораторній роботі №3, зокрема для створення абстрактного класу Trigger та його класів-нащадків (RSTrigger, JKTrigger тощо).
2. **Вимоги до GUI:** Unity надає вбудовану систему **Unity UI**, яка дозволяє швидко та ефективно реалізувати весь необхідний графічний інтерфейс користувача (головне меню, екрани введення даних та відображення результатів).
3. **Нефункціональна вимога (Сумісність):** Однією з ключових нефункціональних вимог (TCN_03) була сумісність ПЗ з основними

операційними системами (Windows, Linux, macOS). Unity є провідним кросплатформеним рушієм, що дозволяє виконати цю вимогу, зібравши проєкт для всіх цільових платформ з єдиної кодової бази.

Таким чином, обрані засоби розробки повністю відповідають задачам проєкту «TriggerCalc» і дозволяють ефективно реалізувати як логіку обчислень, так і вимоги до інтерфейсу та сумісності.

4.

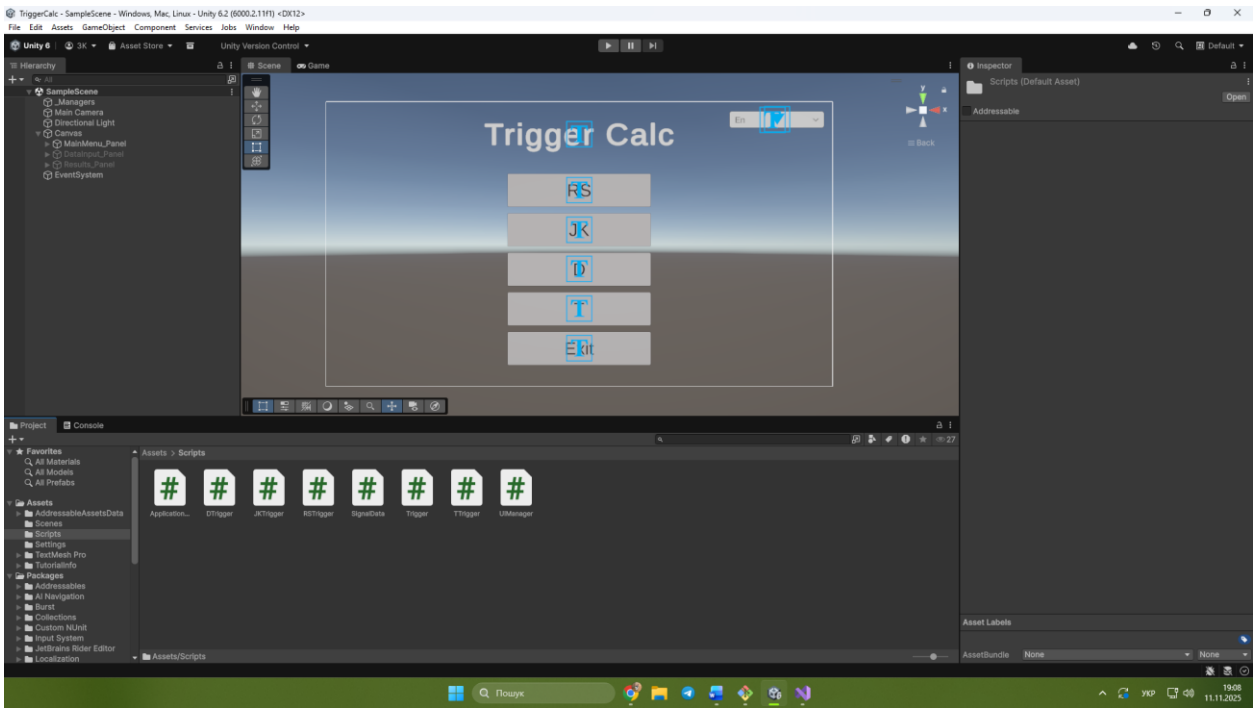

Рисунок 4 – Скриншот з середовища розробки

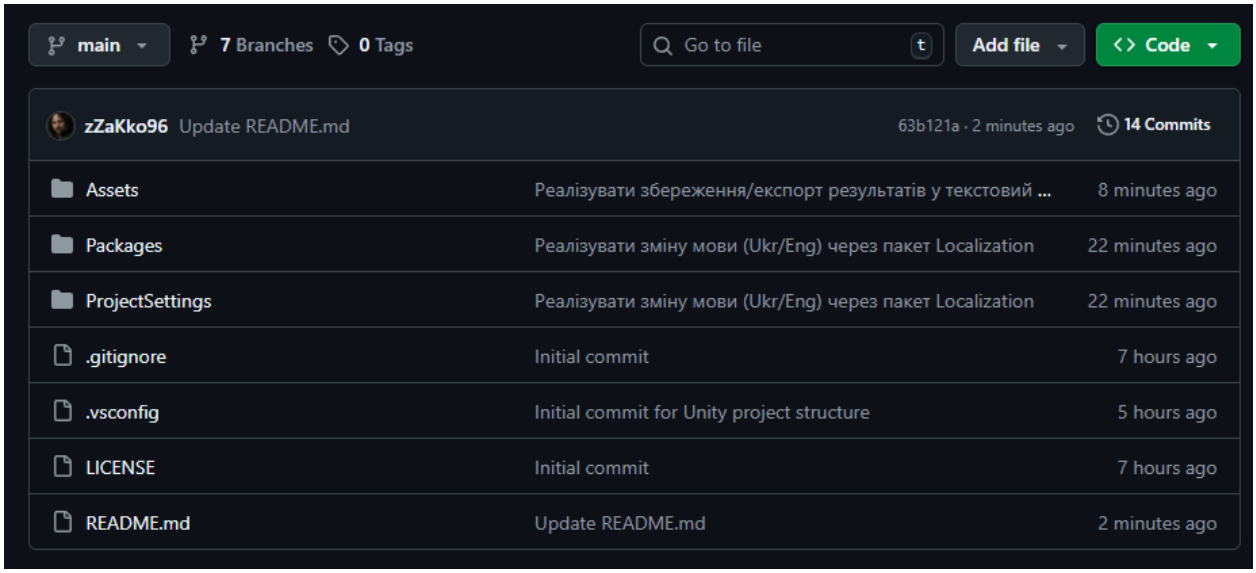

Рисунок 5 – Скриншот репозиторія після завершення роботи

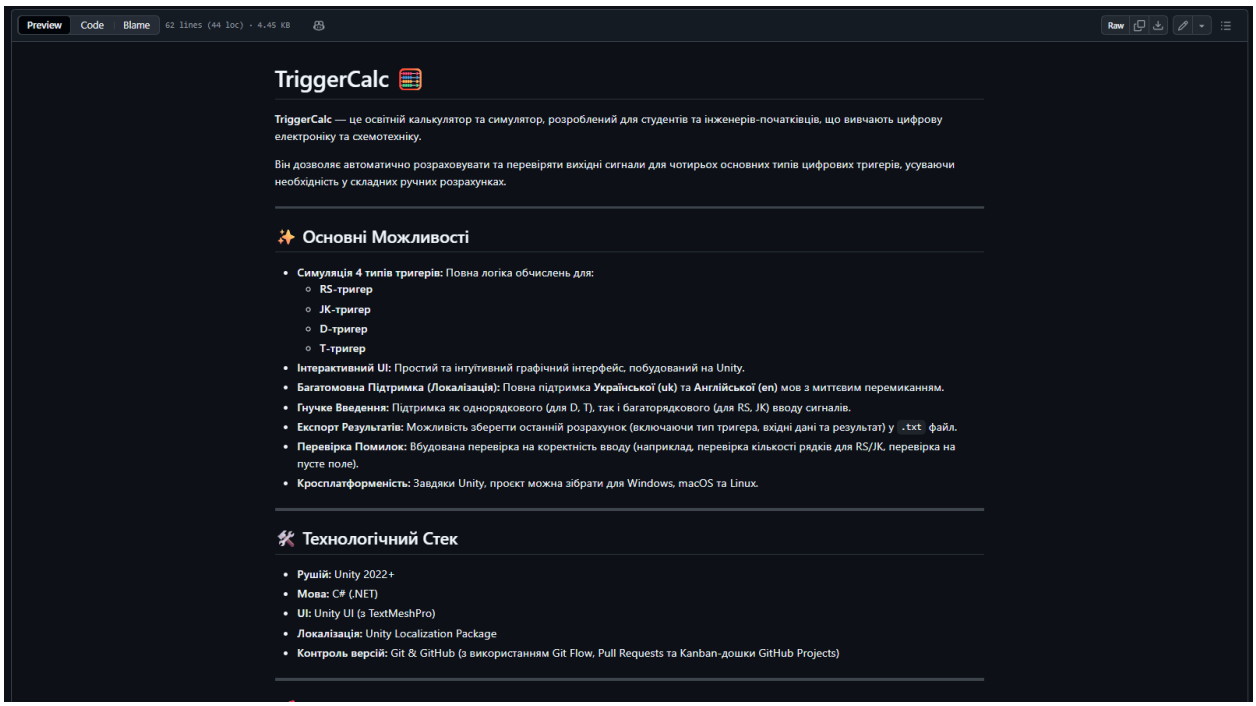

Рисунок 6 – Readme.md

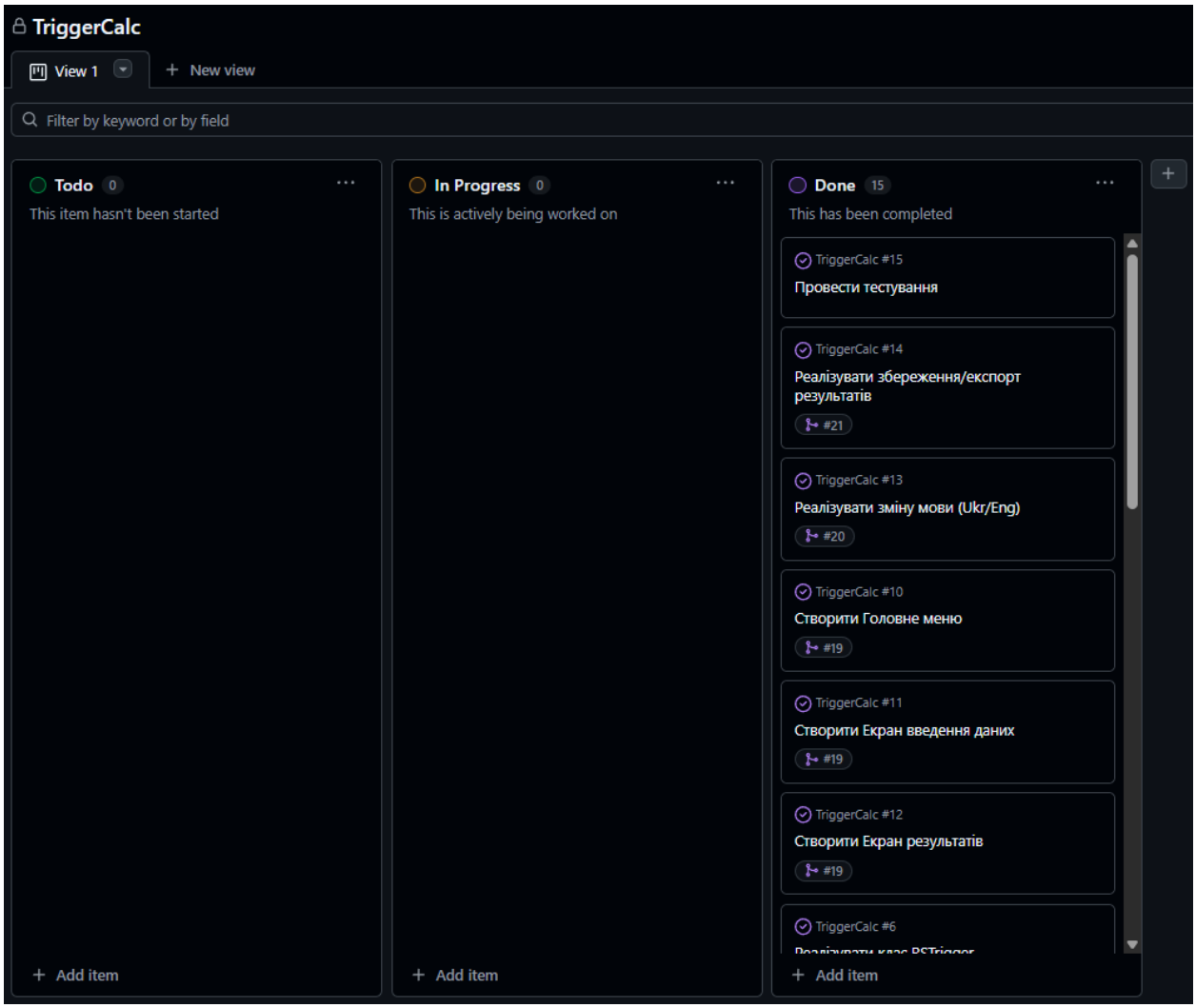

Рисунок 7 – Kanban дошка після завершення роботи

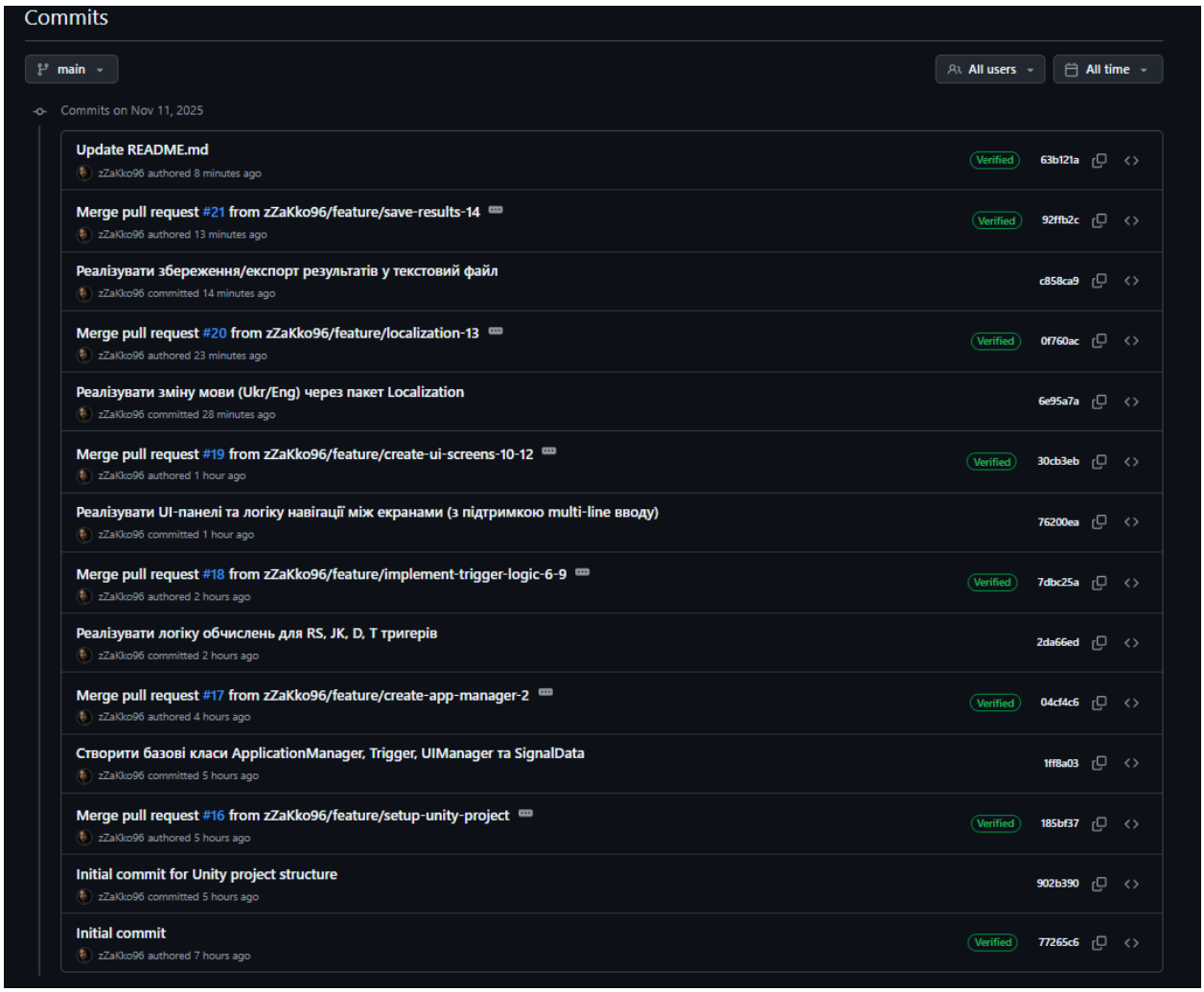

Рисунок 8 – Історія комітів на GitHub

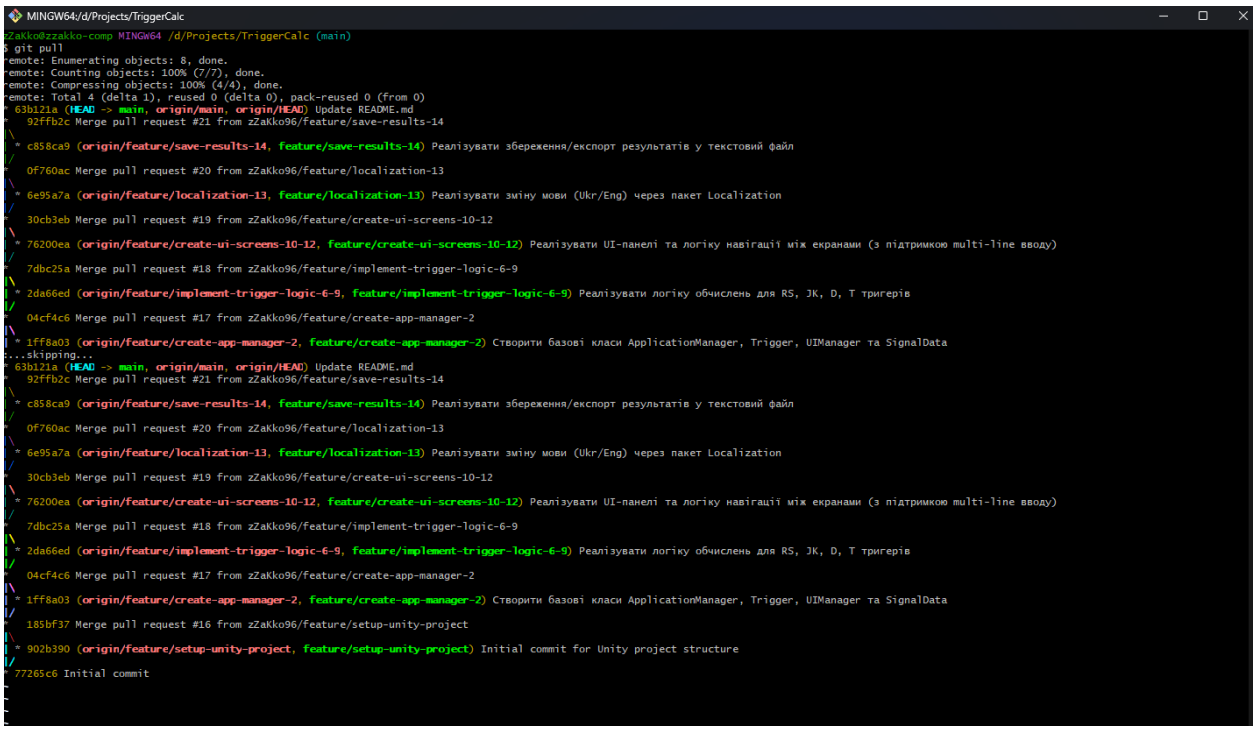

Рисунок 9 – Історія комітів в Git Bash

Вихідний код завеликий, щоб повністю вмістити в звіт. Повністю його можна переглянути в репозиторії Assets/Scripts. Ось невеликі сніпети коду:

Логіка одного тригера:

JKTrigger.cs

```
using UnityEngine;

public class JKTrigger : Trigger
{
    public override void SetInitialState(bool state)
    {
        this.currentState = state;
    }

    public override SignalData CalculateOutput(SignalData input)
    {
        int length = input.inputSequence1.Length;
        input.outputSequence = new bool[length];

        bool[] J = input.inputSequence1;
        bool[] K = input.inputSequence2;

        for (int i = 0; i < length; i++)
        {
            if (J[i] && K[i])
            {
                currentState = !currentState;
            }
            else if (J[i])
            {
                currentState = true;
            }
            else if (K[i])
            {
                currentState = false;
            }
            else
            {
            }

            input.outputSequence[i] = currentState;
        }

        return input;
    }
}
```

Логіка парсингу в UIManager.cs:

```
public void OnButton_Calculate_Click()
{
    string inputText = inputField_1.text;
    SignalData inputData = new SignalData();

    string[] lines = inputText.Split(new[] { '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);

    if (currentTriggerType == "RS" || currentTriggerType == "JK")
    {
        if (lines.Length < 2)
        {
            ShowError("error_rsjk_lines");
            return;
```

```
            }
            if (lines[0].Trim().Length != lines[1].Trim().Length)
            {
                ShowError("error_length_mismatch");
                return;
            }

            inputData.inputSequence1 = ConvertStringToBoolArray(lines[0].Trim());
            inputData.inputSequence2 = ConvertStringToBoolArray(lines[1].Trim());
        }
        else
        {
            if (lines.Length < 1)
            {
                ShowError("error_empty_input");
                return;
            }
            inputData.inputSequence1 = ConvertStringToBoolArray(lines[0].Trim());
            inputData.inputSequence2 = new bool[lines[0].Trim().Length];
        }

        appManager.PerformCalculation(inputData);
    }
```

Виконання завдання 4 стало ключовим етапом, де проєкт «TriggerCalc» перейшов від проєктування (Лабораторна робота №3) до повної програмної реалізації. Робота велася з суворим дотриманням професійного робочого процесу, що поєднував **GitHub Projects (Kanban)** для управління завданнями та **Git** для контролю версій.

Для кожної картки завдань з початкового беклогу (Рисунок 1) виконувався чіткий ітеративний цикл:

1. Картка завдання (наприклад, #13 Реалізувати зміну мови) переміщалася з "Todo" у **"In Progress"** на Kanban-дошці.
2. За допомогою **Git Bash** створювалася нова гілка (branch) від main для ізольованої розробки цієї функції (напр., feature/localization-13).
3. Безпосередньо у середовищі розробки **Unity** (Рисунок 4) писався необхідний C# код, створювалися UI-елементи та налаштовувалася сцена.
4. Після завершення роботи над функцією зміни фіксувалися (commited) та відправлялися (pushed) у віддалений репозиторій на GitHub.
5. На GitHub створювався **"Pull Request" (PR)** для злиття нової гілки (feature/localization-13) назад у main.
6. В описі Pull Request обов'язково додавалося ключове слово (напр., Closes #13), яке пов'язувало PR із відповідною "Issue" (карткою) на Kanban-дошці.
7. Після "Code Review" (у цьому випадку — самоперевірки) Pull Request зливався (merged) у гілку main.

Завдяки зв'язку Closes #13, картка на Kanban-дошці **автоматично** переміщалася у колонку **"Done"**.

Цей цикл був повторений для всіх 15 завдань, від налаштування проєкту до фінального тестування. Результатом є повністю функціональний додаток, чистий репозиторій (Рисунок 5), детальний README.md (Рисунок 6), завершена Kanban-дошка (Рисунок 7) та професійна, легко читома історія комітів, яка демонструє злиття всіх гілок функцій (Рисунок 8 , Рисунок 9).

Висновки

В ході виконання лабораторної роботи №4 було успішно завершено етап програмної реалізації проєкту «TriggerCalc», що дозволило отримати практичні навички застосування сучасних інструментів для створення та зберігання вихідного коду.

Під час роботи було виконано такі ключові завдання:

1. **Налаштовано управління проєктом:** За допомогою **GitHub Projects** було створено **Kanban-дошку**, на якій весь проєкт було декомпозовано на 15 окремих завдань (Issues). Це дозволило візуалізувати беклог та відстежувати прогрес виконання.
2. **Створено та налаштовано репозиторій:** На платформі **GitHub** було ініціалізовано репозиторій TriggerCalc, до якого одразу було додано README.md, ліцензію MIT та .gitignore для Unity, забезпечивши чисту та професійну структуру проєкту.
3. **Реалізовано програмний продукт:** Використовуючи обраний стек (C# та Unity), було створено повноцінний додаток, який реалізує весь запланований функціонал: від логіки обчислень тригерів до багатомовного графічного інтерфейсу та експорту результатів.
4. **Опановано робочий процес Git Flow:** Вся розробка велася з використанням системи контролю версій **Git**. Кожна нова функція (Issue) реалізовувалася в окремій **гілці (branch)**, а її додавання до основного коду відбувалося через **Pull Requests (PR)**. Це забезпечило чітку, керовану та професійну історію комітів.

Таким чином, мета лабораторної роботи була досягнута. Я здобув практичний досвід у повному циклі розробки: від планування завдань на Kanban-дошці до реалізації коду, використання гілок Git та проведення "Code Review" через Pull Requests.