

Міністерство освіти і науки України
Черкаський національний університет імені Богдана Хмельницького
Факультет обчислювальної, інтелектуальних та управляючих систем
Кафедра програмного забезпечення автоматизованих систем

КУРСОВА РОБОТА З ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ на тему «Ігровий застосунок “2 3 4 Player Games”»

Студента 2 курсу, групи КС-231
спеціальності 121 «Інженерія
програмного забезпечення»
Філіпенко А.М.

(прізвище та ініціали)

Керівник _____
доцент, к.т.н. Супруненко О.О.

(посада, вчене звання, науковий ступінь,
прізвище та ініціали)

Оцінка за шкалою:

(національною, кількість балів, ECTS)

Члени комісії

(підпис) (прізвище та ініціали)

(підпис) (прізвище та ініціали)

(підпис) (прізвище та ініціали)

ЗМІСТ

Table of Contents

ВСТУП	2
РОЗДІЛ 1. ПРОЄКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ «2 3 4 Player Games». БАЗОВІ ПРИНЦИПИ ТА ПОБУДОВА	3
1.1 Вибір платформи для побудови ігрового застосунку	3
1.2 Порівняльний аналіз ігрових застосунків	4
1.3 Принципи ООП	4
1.4 Алгоритм взаємодії об'єктів між собою	5
1.5 UML-діаграми.....	6
1.6 Векторна математика руху та інше	7
Висновки	9
РОЗДІЛ 2. ПРОЄКТУВАННЯ ПРОГРАМНОГО МОДУЛЯ ДЛЯ ІГРОВОГО ЗАСТОСУНКУ «TANKS»	11
2.1 Аналіз вимог	11
2.2 Блок-схема запуску гри.....	12
Висновки	14
РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ «2 3 4 Player Games»	16
3.1 Реалізація функцій меню	16
3.2 Реалізація ООП	24
Висновки	25
ВИСНОВОК	26
Список інформаційних джерел	29
Додаток А. Лістинг коду алгоритму «Pixel perfect collision detection»	30
Додаток Б. UML-діаграма ігрового застосунку «Tanks».....	31
Додаток В. Блок-схема роботи методу Start() класу Tournament.....	32

ВСТУП

Геймдев наразі є однією з найбільш динамічною та інноваційною галуззю у сучасній сфері програмування, що постійно розвивається, та притягує охочих працювати в ній та вкладати свій вклад. Створення будь-якого ігрового застосунку включає необхідність використання об'єктно-орієнтованого програмування задля задання правильної ієрархії класів, побудови зв'язків. Тому дана тема роботи чудово може показати знання використання ООП у цій сфері.

Метою даної роботи є створення ігрового застосунку, з використанням принципів ООП, демонстрації та застосування принципів створення 2D-ігор з використанням допоміжної бібліотеки SFML у мові програмування C#. Для цього поставлено завдання:

1. Показати вміння використання напрямів ООП.
2. Створення вигідних алгоритмів для побудови ігрового застосунку, руху об'єктів, взаємодії між ними.
3. Практично освоїти навички побудови простої 2D гри.

РОЗДІЛ 1. ПРОЄКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ «2 3 4 Player Games». БАЗОВІ ПРИНЦИПИ ТА ПОБУДОВА

1.1 Вибір платформи для побудови ігрового застосунку

Для побудови простих ігор існує багато бібліотек, застосунків та платформ, тому треба перед роботою обрати ту, яка найбільше підходить. На вибір розглядались дві популярні кросплатформні фреймворки/бібліотеки – Monogame [1] та SFML[2]. Вони використовуються саме для розробки ігор, особливо 2D ігор, і використовують мову C# для написання коду. Наведемо основні недоліки та переваги їх обох.

Переваги Monogame:

- є перевиданням фреймворку XNA Framework, тому для розробників, що вже мали з ним роботу, Monogame є дуже легким для розуміння написання коду;
- кросплатформність;
- має власну систему обробки контенту під формат гри;

Недоліки Monogame:

- відсутність вбудованого редактора, як Unity чи Godot;
- для новачків цей фреймворк може здатись заскладним;
- потенційні проблеми з продуктивністю.

Переваги SFML:

- простота та високий рівень контролю над усіма графічними та візуальними операціями;
- модульність, що дозволяє працювати з одним напрямом (аудіо, графіка, мережа, тощо) незалежно від один одного;
- архітектуру гри користувач сам створює, просто використовуючи вбудовані методи розробки
- простий для використання новачкам.

Недоліки SFML:

- також відсутність вбудованого редактора;
- більше ручної роботи над побудовою структури програми;
- є деякі обмеження на деяких платформах.

Отже, після надання переваг та недоліків обох бібліотек, було обрано використання саме бібліотеки SFML. Він є зручним для початківців, що тільки починають свою кар'єру у геймдеві, особливо у розробці 2D ігор. Це дозволить самостійно мати високий контроль над кодом, пишучи власні методи використовуючи уже наявні, з використанням графіки, системи і т.д.

1.2 Порівняльний аналіз ігрових застосунків

Назва теми була взята уже з існуючого ігрового застосунку, доволі популярного серед молоді – «2 3 4 Player games»[3]. Сама ж гра складається з деяких міні ігор, які вибираються для створення турніру відразу для 2-4 гравців.

Було зроблено вибір саме для створення десктопного варіанту гри, так як вона сама є мобільною, з реалізацією одної міні гри «Tanks». У ній користувачі керують своїми танками для знищення один одного. Це дозволило використати в майбутній реалізації векторної алгебри, алгоритми колізії між танками та боєприпасами суперника.

Схожі ігри на тематику танків уже створювались користувачами, використовуючи SFML [4]. Даний застосунок також писався під ООП, проте використовуючи мову C++.

1.3 Принципи ООП

Курсова робота з ООП охоплює у собі використання всіх головних принципів ООП, таких як поліморфізм, інкапсуляція, успадковування та абстракція. Задля розуміння як використовуються вони у роботі, наведемо їх тлумачення [5]:

1. Поліморфізм вміщує в собі використання одного інтерфейсу чи деякої структури та перевизначення цієї концепції під різні типи задач.

Розглядають такі типи як спеціалізований, параметричний та поліформізм типів. У застосунку буде використовуватись спеціалізований поліморфізм, який перевизначає загально створені методи під конкретний тип задачі для класу.

2. Інкапсуляція – це поняття вміщує у себе приховування деталей реалізації полів, методів та інших об’єктів між різними класами. Тобто, деякі класи можуть мати доступ до деяких полів, використовувати їх, а вже інший клас цієї можливості не має. У застосунку буде використовуватись такі параметри інкапсуляції як `private` (поля та методи використовуються лише у класі, в якому вони створені; для доступу створюються метод доступу і(або) метод-мута́тор), `public` (до об’єкта можна на пряму звертатись, нема жодної заборони) та `protected` (поля та методи доступні лише всередині базового та у дочірніх класах).
3. Успадковування – концепція, яка дозволяє створювати базовий клас загального призначення, що вміщує в собі загальні поняття, аби різні дочірні класи могли успадковувати їх. Як наслідок, нам не потрібно писати у дочірніх класах поняття, що є однаковими, тому задля скорочення коду можна прописати їх у батьківському.
4. Абстракція дозволяє нам спрощувати складні системи, зосереджуючись лише на їхніх характеристиках, приховуючи непотрібні деталі реалізації. Одним словом, ви вказуємо лише «що» об’єкт робить, але не «яким чином».

1.4 Алгоритм взаємодії об’єктів між собою

У ігровому застосунку аби отримати правильні колізії з реальними формами об’єкта, було використано не просту перевірку меж спрайтів, у вигляді прямокутників (Bounding Box Collision), а попиксельною перевіркою для спрайтів (Pixel Perfect Collision Detection) [6].

Цей алгоритм дає високу точність , адже враховує реальні форми об’єктів з усіма можливими кутами та формами, а не просто перевіряє прямокутник, що

обмежує об'єкт. Через це ми не спостерігаємо порожніх колізій, коли два спрайти візуально не торкаються, проте прямокутники перетинаються.

Проте цей алгоритм є доволі громіздким, адже перевіряє кожен піксель на місці перетину спрайтів. Тому перш за все виконується груба перевірка на перетин самих прямокутників, а вже потім сам етап перевірки.

1.5 UML-діаграми

Для будь-якого проєкту, що використовує загальні принципи об'єктно-орієнтованого програмування потрібно добре розуміти як його класи поєднуються між собою, у яких зв'язках вони перебувають, що кожен клас імплементує чи успадковує. Для цього існує доволі зручний інструмент для відображення – UML-діаграма. Через неї розробник може показати всі основні сполучення об'єктів між собою у доволі лаконічному вигляді. Для створення UML було використано застосунок UMLet [3].

Розглянемо основні візуальні характеристики діаграми:

1. Роздільники секцій у класах: Fields, Properties, Methods. Це не є обов'язковим, проте для читабельності було додано.
2. {static},{const} і {readonly} – відповідно показують статичні та незмінні поля.
3. Ідентифікація зв'язків – це зручно показувати різними видами стрілок, що позначають свій власний тип взаємодії (рис. 1.5.1).

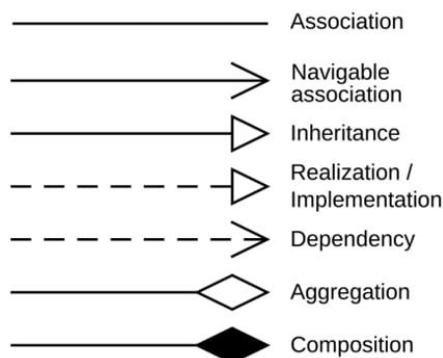


Рисунок 1.5.1. Типи міжкласових зв'язків

Деякі з цих зв'язків ми ще не розглядали наведемо короткий опис:

1. Асоціація – загальний тип для всіх зв'язків, коли між двома класами існує деяке смислове відношення, наприклад коли один клас має посилення на інший у вигляді поля, чи через взаємодії у методах. Може бути двонаправленою – обидва класи мають посилення на один одного, чи однонаправленою – лиш один з класів має посилення на інший.
2. Навігаційна асоціація – та ж одностороння асоціація.
3. Успадковування – успадковування класів.
4. Реалізація/Імплементация – реалізація інтерфейсу.
5. Залежність – один клас тимчасово використовує інший у вигляді локальної змінної, параметр входу. Одним словом, він не є його постійним членом.
6. Агрегація – вид асоціації, при якій деяка «частина» може існувати незалежно від «цілого»
7. Композиція – протилежна агрегації, «частина» не може жити без «цілого».

1.6 Векторна математика руху та інше

Задля забезпечення правильної логіки руху об'єктів у застосунку потрібно використовувати базові принципи побудови векторів з урахуванням обертання об'єкта.

Для обчислення напрямку руху об'єктів необхідно вирахувати відповідний кут для обчислення вектора руху. Проте, перед цим ми маємо представити кути у радіанах, для цього використовується формула 1.6.1:

$$\alpha = \alpha^{\circ} \cdot \frac{\pi}{180} \quad (1.6.1)$$

α – кут у радіанах;

α° – кут у градусах.

Після того як обчислено кут, ми можемо вирахувати уже сам вектор (формула 1.6.2):

$$u = (\cos(\alpha), \sin(\alpha)) \quad (1.6.2)$$

u – вектор напрямку;

$\cos(\alpha)$ – X-складова вектору;

$\sin(\alpha)$ – Y-складова вектору.

Після цього ми множимо отриманий вектор на швидкість і наш об'єкт буде відповідно у цьому напрямку рухатись.

Тепер перейдемо до колізій. Коли об'єкт-танк колізує з якимсь об'єктом, ми маємо заборонити рух танка у цьому напрямку і відштовхнути їх звідти. Отже, тут потрібні три умови:

1. Обчислюється вектор різниці позицій (1.6.3):

$$D = P_1 - P_2 \quad (1.6.3)$$

D – вектор різниці;

P_1 – позиція першого об'єкта;

P_2 – позиція першого об'єкта.

2. Обчислення довжини вектора різниці (1.6.4):

$$|D| = \sqrt{D_x^2 + D_y^2} \quad (1.6.4)$$

$|D|$ – довжина вектор різниці;

D_x – X-складова вектора різниці;

D_y – Y-складова вектора різниці.

3. Коли ми отримали нашу довжину вектора різниці, перевіряємо на рівність нулю. Якщо так знаходимо вектор відштовхування (1.6.5):

$$P'_1 = P_1 + \frac{D}{|D|} S_{\text{відштовх}} \quad (1.6.5)$$

P'_1 – новий вектор першого об'єкта;

$S_{\text{відштовх}}$ – константа рівна δf .

Отже, саме через ці формули ми можемо забезпечити правильне переміщення об'єктів під час колізії.

У застосунку, я також використав обгортання екрану (Screen Wrapping). Цей алгоритм дозволяє об'єктам виходячи за межі екрана з'являтися на протилежному боці. Для коректної обробки, потрібно перевіряти X та Y координату об'єкта на перетин з сторонами екрану (1.6.6 і 1.6.7)

$$P'_x = (P_x + W) \bmod W \quad (1.6.6)$$

$$P'_y = (P_y + H) \bmod H \quad (1.6.7)$$

P'_x – нова X-координата об'єкта;

P'_y – нова Y-координата об'єкта;

P_x та P_y – поточні координати об'єкта;

W – ширина екрану.

H – висота екрану.

Додавання W(H) є важливим кроком для отримання позитивної відповіді. Так як частка від ділення може бути від'ємною, без додавання об'єкт не з'явиться з протилежної сторони і буде за межами видимої області.

Висновки

У першому розділі ми розглянули основи побудови нашого застосунку та які уже існуючі принципи будемо використовувати у проєкті. Підсумуємо:

1. Обрали платформу SFML для побудови ігрового застоснку, надали опис її обґрунтували її вибір.
2. Обґрунтували тему курсової, адже вона була взята на основі уже існуючої мобільної гри та надали схожий вже проєкт нашому.
3. Надали основні принципи об'єктно-орієнтованого програмування, які ми маємо використати у проєкті.
4. Надали принципи побудови UML-діаграми.
5. Надали інформацію про алгоритми, що будуть використовуватись у проєкті.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ПРОГРАМНОГО МОДУЛЯ ДЛЯ ІГРОВОГО ЗАСТОСУНКУ «TANKS»

2.1 Аналіз вимог

Перш ніж спроектувати правильну структуру свого ігрового застосунку, необхідно проаналізувати вимоги до нього. Цей етап є початковим та основним, адже перед самим створенням програми потрібно розуміти, що саме потрібно у ньому створити, чи це буде корисно та зручно користувачам. Тому для застосунку «2 3 4 Player Games» було окреслено такі основні вимоги:

1. Гра повинна забезпечити покроковий процес застосунку відповідно побажань користувачів. Тому для цього я створив зручний ігровий цикл, що дозволяє 2-4 користувачам грати проти один одного.
2. Система повинна запускати ігровий процес відповідно обраної кількості гравців. Для цього у вікні меню я прописав відповідні кнопки, незалежно від обраної з такою кількістю танків і запуститься гра.
3. Кожен з гравців не має заважати іншим гравцям під час гри керувати своїми об'єктами, тому розмістив так кнопки взаємодії, аби не заважали один одному.
4. Гра має забезпечити правильний аналіз підрахунку балів, визначення знищених та ще живих танків на ігровому полі.
5. Гра має відслідкувати переможця всіх раундів, незалежно чи перерваний був сам турнір, чи він був проведений до кінця.

Інтерфейс та навігація має надавати функції згідно заданих умов:

1. Головне меню з вибором функціоналу: кнопки вибору кількості гравців, перегляду ігрових сеансів та інформації як грати.
2. Після вибору кількості гравців запустити ігровий процес.
3. Вивести фінальні результати.

Надамо також словник термінів нашої програми аби не було якихось непорозумінь у подальшому використанні (табл. 1):

Таблиця 1.

Танк	Основний ігровий об'єкт, саме ним користувач грає
Бомба	Знаряддя знищення опонента
Коробка-сюрприз	У цих коробках є різні додаткові сили для танків. Їх дві: щит та зменшення танка у розмірах
Блок-стіна	Текстуру на мапі, крізь яку об'єкти не проходять. За ними танк може заховатись від опонентів.

2.2 Блок-схема запуску гри

Тепер скористаємось таким інструментом як блок-схема. Ця схема давно всім відома і є базовою навичкою для кожного програміста. Завдяки ній можна показати простими схемами будь-який алгоритм, від загальної, спрощеної логіки, до продеталізованої роботи будь-якого методу. Блок-схеми дуже є зручними для початківців для візуальної роботи своєї майбутньої програми.

Тому зобразимо роботу гри, почнімо з того кроку, коли вже користувачі вибрали скільки гравців було обрано, і це значення буде передаватись програмою на подальшу роботу програми (рис. 2.3.1).

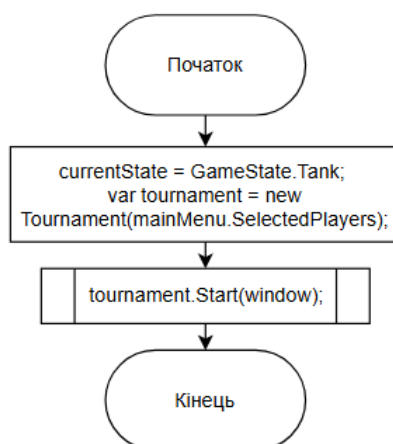


Рисунок 2.3.1. Запуск турніру

Отже, після того як ми отримали кількість гравців у головному класі Program(), запускається метод Start() класу Tournament. У ньому, виконується логіка запуску турніру з трьох сесій, кожна з них має власне оформлення карти, контролюється цикл турніру, тобто передбачено в ньому аварійний вихід з турніру. Сама блок-схема цього методу розміщена у додатку В.

Після того як запустилась ігрова сесія, створюється екземпляр класу TankGame, що є ключовим в управлінні ігрового процесу. Він відповідає за:

1. Створення та конфігурації карти, за генерування блоків-стін, коробків-сюрпризів, забезпечення, аби танки не спавнились у вже зайнятих місцях.
2. Завантажує та розподіляє усі ресурси відповідно до кольору об'єктів (червоний, синій, зелений, жовтий).
3. Управляє візуалізацією програми, адже саме через його методи у класі ігрової сесії коректно відображаються карта та об'єкти.

Для більш глибокого розуміння як обробляється рух танків та бомб, ігрових об'єктів, також надам блок-схему (рис. 2.3.2).

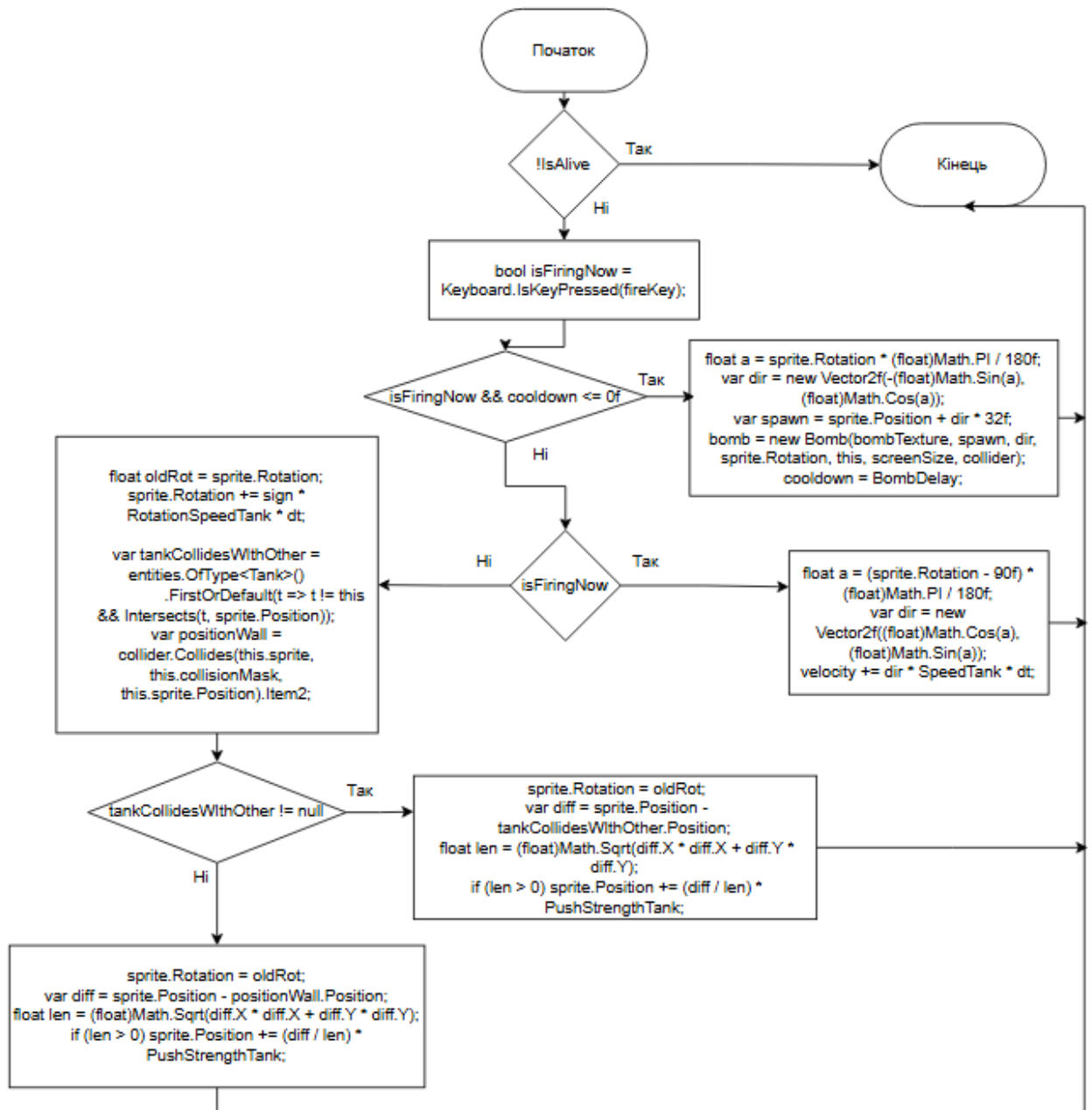


Рисунок 2.3.2. Метод HandleInput() класу Tank.

Висновки

Цей розділ був присвячений комплексному проектуванню ігрового застосунку. «Tanks», що є обов'язковим та основним етапом перед тим, як писати сам код. У першому підрозділі ми проаналізували, які саме потреби та

функціонал нам необхідно написати у нашому застосунку. Відповідно до цих умов було прописано код ігрового застосунку.

У третьому модулі я прописав блок-схеми, що детально показали як працюють програмні алгоритми. Як проводяться турніри ігор, ігрові сесії, як рухаються об'єкти та інше. Якщо підсумувати вище зазначене, у другому модулі я показав архітектури програми . Такий підхід безпосередньо заклав майбутню реалізацію продукту.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ «2 3 4 Player Games»

3.1 Реалізація функцій меню

Розглянемо будову та функціонал першого вікна, яке користувачеві висвітлиться – меню. Меню передбачає використання маніпулятора «миші» для виконання вибору кнопки. Тобто необхідно навести мишку на бажану кнопку, та правою кнопкою миші натиснути. Відповідно колір буде змінено, що означатиме вибір та наведення на кнопку.

У меню було реалізовано 5 кнопок (рис.3.1.1):

1. 2 Players – запускається гра для двох гравців.
2. 3 Players – запускається гра для трьох гравців.
3. 4 Players – запускається гра для чотирьох гравців.
4. View history – запускається вікно для відображень історії сеансів турнірів.
5. How to play – відкривається вікно, у якому розміщено основну інформацію як працює цей ігровий застосунок.

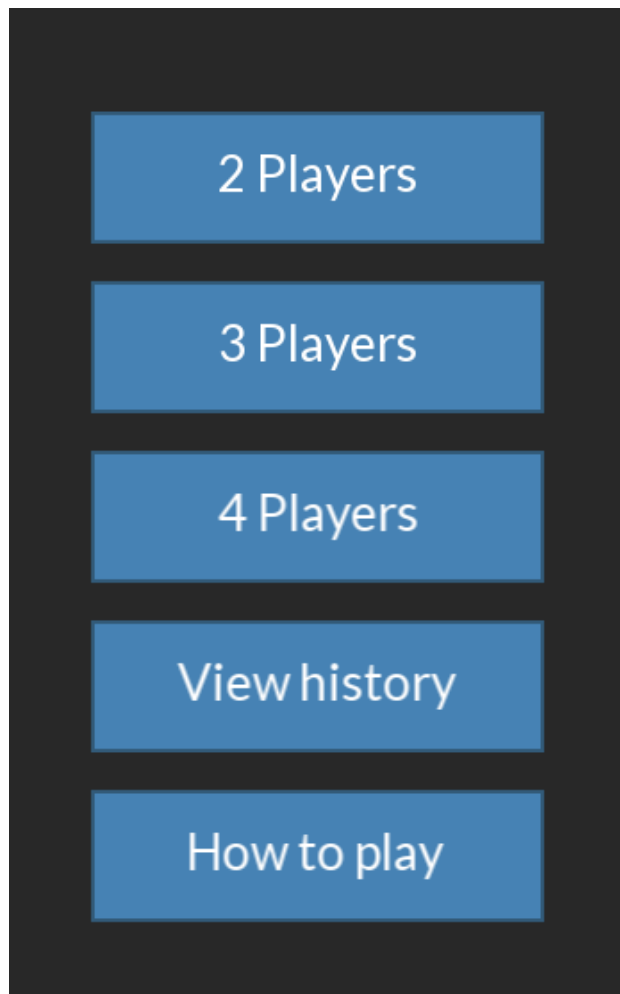


Рисунок 3.1.1. Кнопки основного меню

Тепер перейдемо до функціоналу кожної з кнопок, почнімо з «How to play». Після натиску на неї, відповідно відкриється вікно з текстом-поясненням як грати в гру (рис 3.1.2).

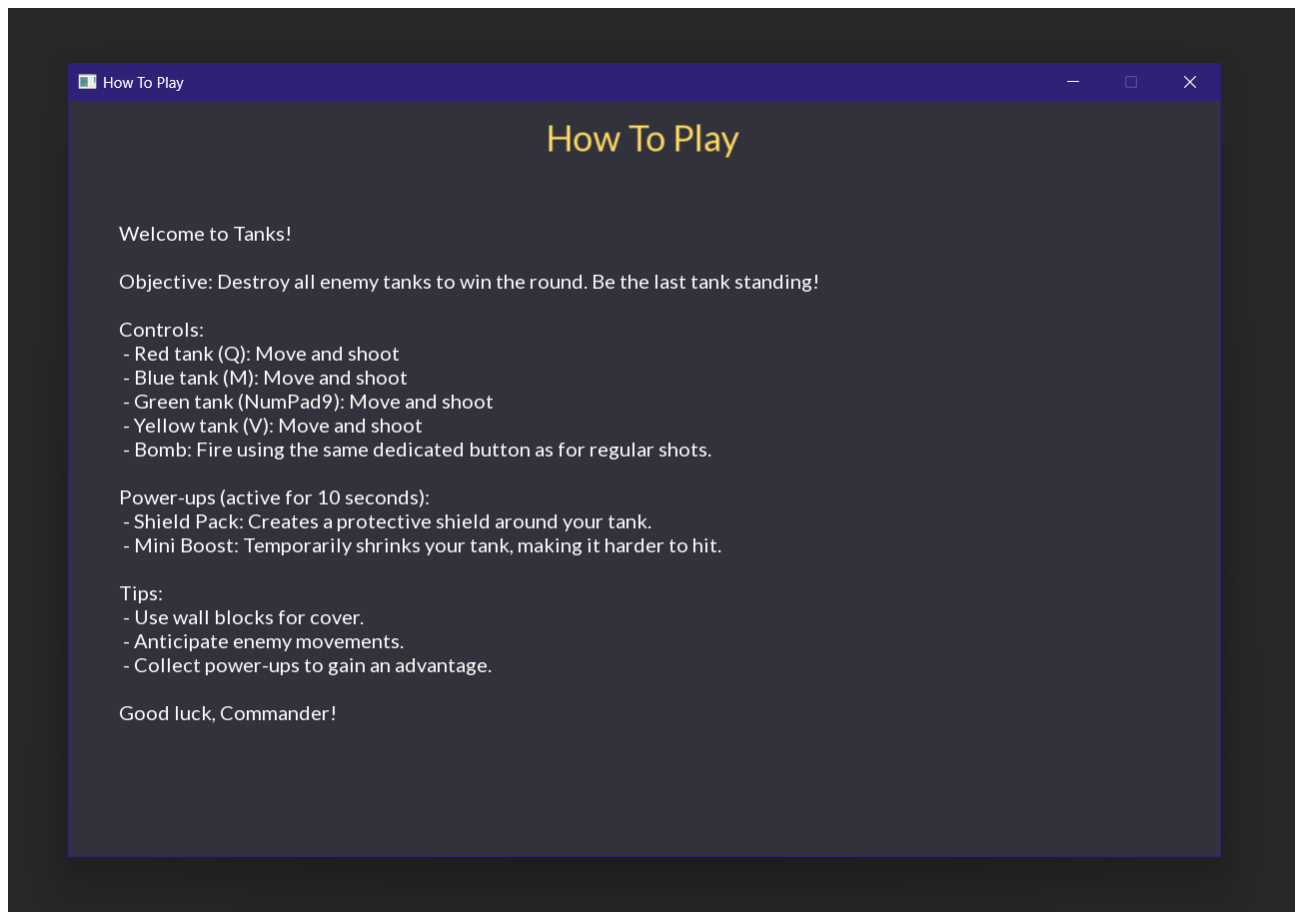


Рисунок 3.1.2. Вікно «How to play»

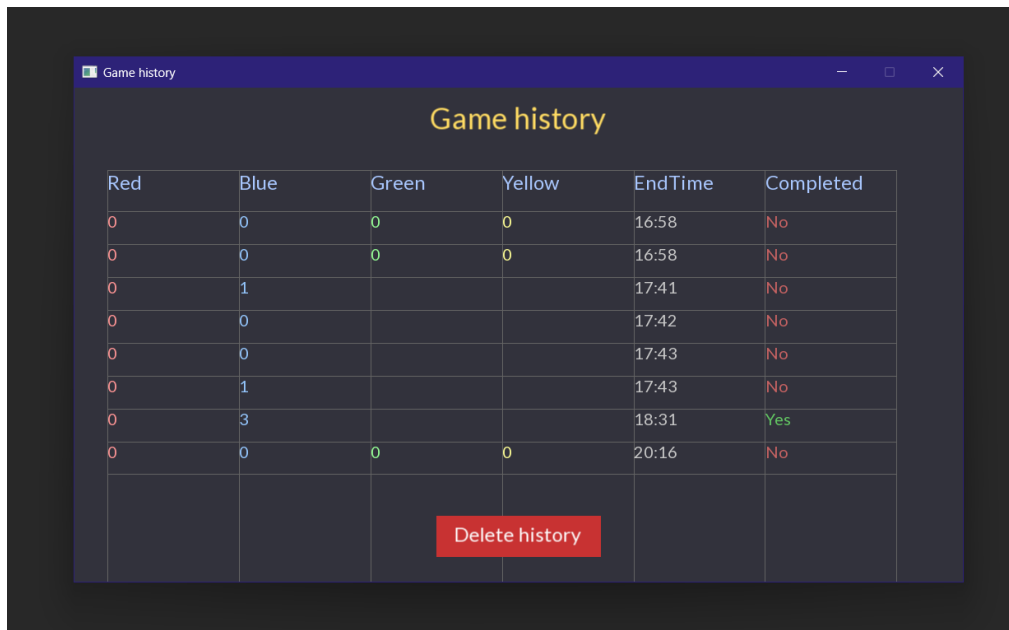
Як бачимо, відображається відповідний текст. Аби закрити вікно і вернутись до основного меню, достатньо натиснути «хрестик».

Тепер перейдемо до кнопки «View history» (рис 3.1.3.). На рисунку зображено схоже вікно, проте вже з таблицею, що показує всі турніри здійснені з моменту останнього очищення даних. Тепер поясню що кожен зі стовпчиків означає:

1. Red – бали червоного гравця.
2. Blue – бали синього гравця.
3. Green – бали зеленого гравця.
4. Yellow – бали жовтого гравця.
5. EndTime – показує час, коли турнір був закінчений.

6. Completed – показує, чи даний турнір був закінчений повністю, чи трапився якийсь аварійний вихід.

Під таблицею є кнопка, функціоналом якої є очищення історії турнірів. Вона очищує відповідний файл, у якому зберігаються дані і опісля висвітлює вікно з текстом «History was removed!» (рис. 3.1.4.). Після цього історія буде очищена.



The screenshot shows a window titled "Game history" with a table of game results. The table has six columns: Red, Blue, Green, Yellow, EndTime, and Completed. There are ten rows of data. A red button labeled "Delete history" is located below the table.

Red	Blue	Green	Yellow	EndTime	Completed
0	0	0	0	16:58	No
0	0	0	0	16:58	No
0	1			17:41	No
0	0			17:42	No
0	0			17:43	No
0	1			17:43	No
0	3			18:31	Yes
0	0	0	0	20:16	No

Delete history

Рисунок 3.1.3. Вікно «Game history»

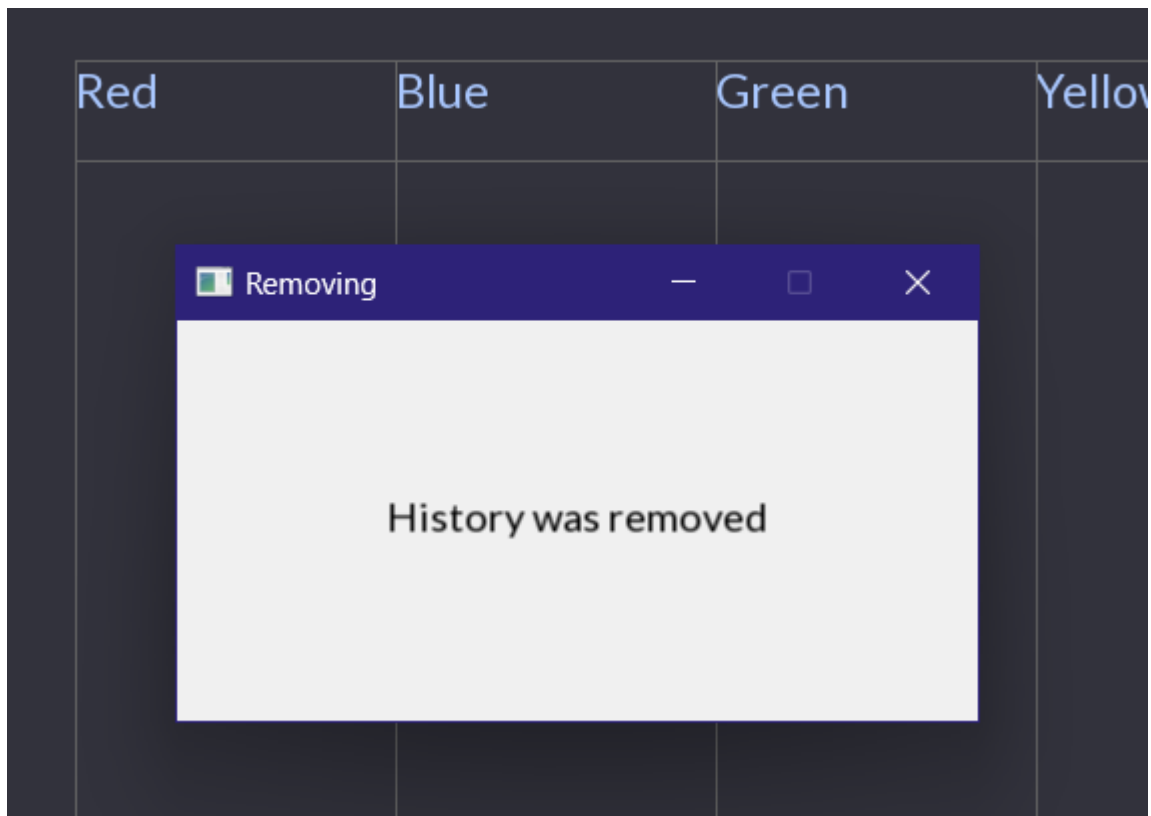


Рисунок 3.1.4. Вікно «Removing»

Далі йде функціонал кнопок з вибором гравців. У них він ідентичний, і запускає гру з відповідною кількістю гравців. Покажу на прикладі 4 гравців (рис. 3.1.5):

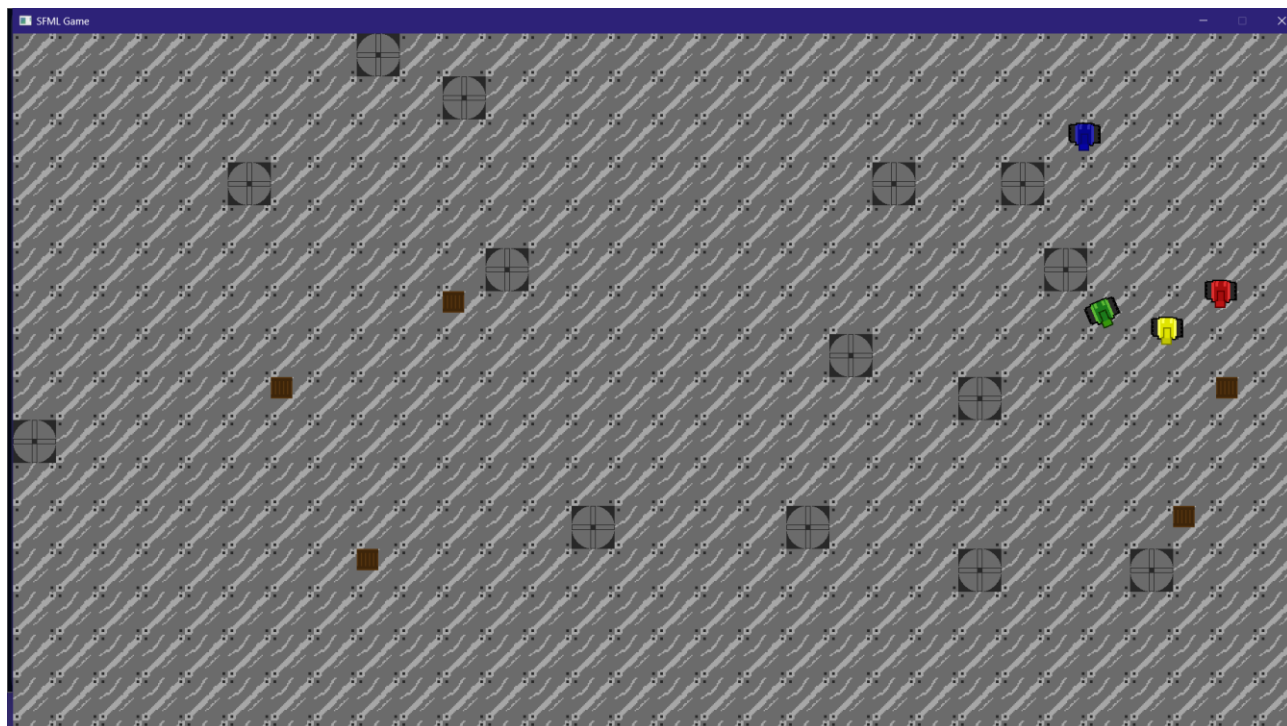
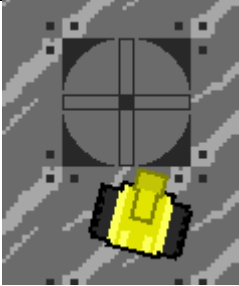
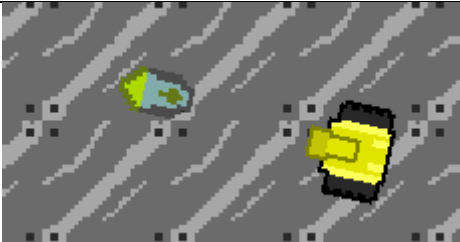


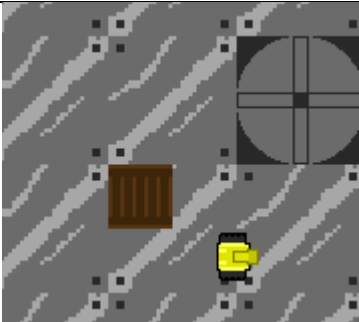
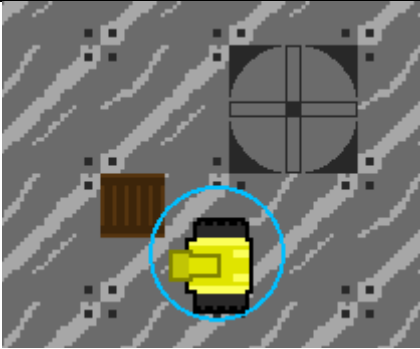

Рисунок 3.1.5.Ігровий турнір для 4 гравців

Тепер покажемо взаємодії всіх об'єктів на карті (табл. 3.1.1.)

Взаємодія об'єктів

Таблиця 3.1.1

Рисунок	Пояснення
1	2
	Танк не може рухатись за блок-стіну
	Танк випускає свою бомбу

1	2
	<p>Танк підібрав один з ящиків і у ньому зменшалась структура.</p>
	<p>Танк підібрав один з ящиків і у нього з'явився щит.</p>
	<p>Знищений танк після колізії з чужою бомбою</p>

Саме такі взаємодії присутні на карті. Тобто саме таким чином гравці грають між собою. Тепер покажемо які ще карти у нас присутні (рис 3.1.6 та рис.3.17).

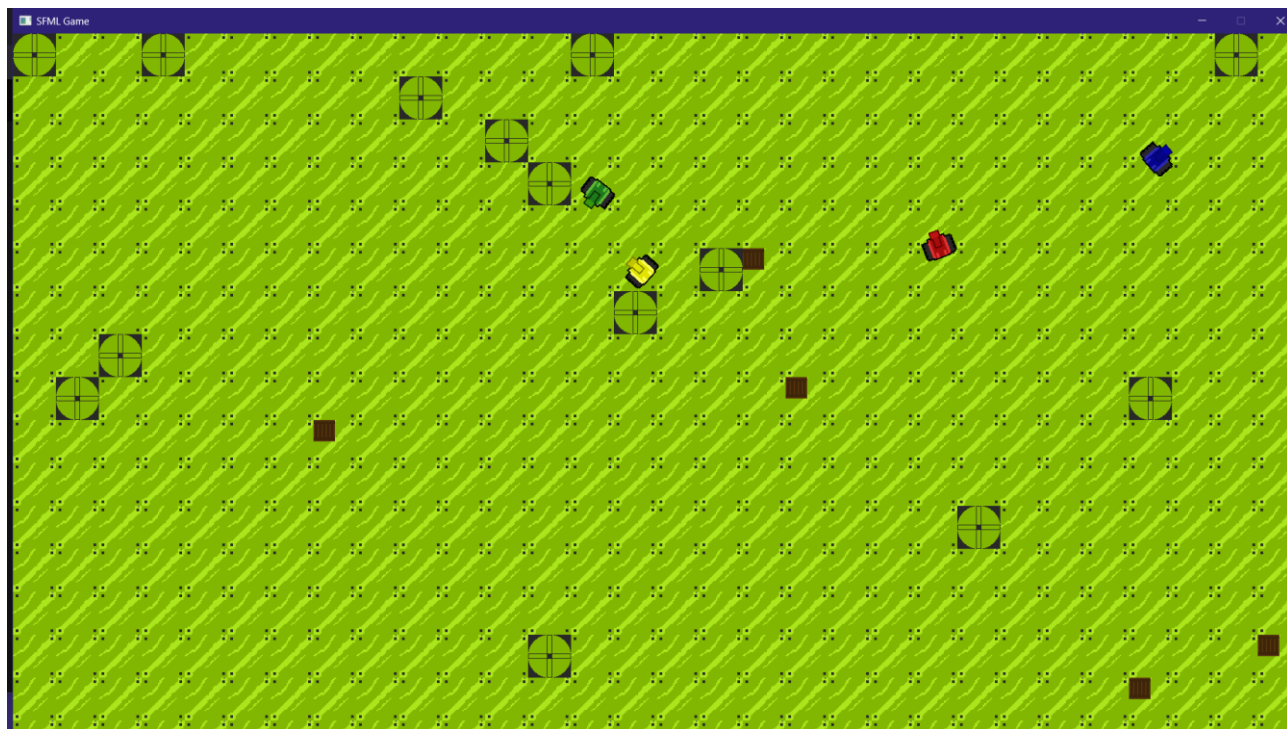


Рисунок 3.1.6. Друга ігрова сесія

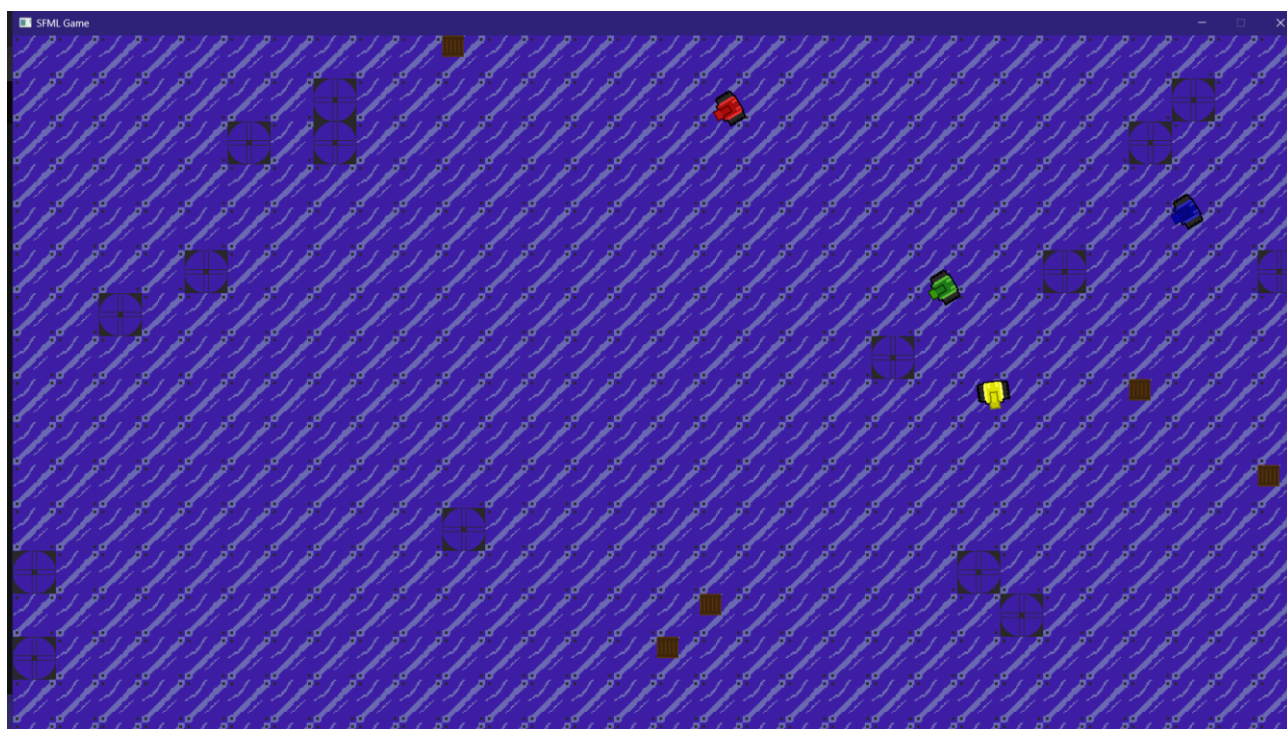


Рисунок 3.1.7. Третя ігрова сесія.

Після того як всі три ігрові сесії було проведено, відповідно висвітлиться турнірна таблиця, в якій показано скільки в кожного гравця було набрано балів (рис 3.1.8.).



Рисунок 3.1.8. Турнірна таблиця

3.2 Реалізація ООП

Перш за все наведемо UML-діаграму (додаток Б). Як ми проговорювали раніше, вона використовується для зображень зв'язків класів у програмі.

1. Абстракція дозволяє приховувати складні деталі реалізації, надаючи лише необхідний інтерфейс для взаємодії.
 - а) Клас `GameEntity` є абстрактним, він визначає обов'язкові поля для ігрових об'єктів `Tank` і `Bomb`. Також має два абстрактних методів `Update()` та `Draw()`, деталі реалізацій яких залишається за похідними класами.
 - б) Клас `Bomb()` та `Draw()` використовують конструктор абстрактного класу, приймаючи значення полів `MapCollider` та `Vector2u`.
2. Поліморфізм дозволяє обробляти об'єкти різних класів через єдиний інтерфейс.

- a) Перевизначення вище наведених методів класу GameEntity дозволяє для кожного об'єкта GameEntity викликати ці методи без зазначення конкретного типу.
 - b) Також сюди відноситься імплементування інтерфейсів, що є у програмі. Кожен клас, що імплементує якийсь інтерфейс, перевизначає його методи.
3. Інкапсуляція активно використовується для приховування внутрішньої реалізації об'єктів.
- a) Усі поля у всіх класах є приватні, і доступ до них з інших класів можна отримати лише через властивості. Це дозволяє нам змінювати їх значення лише в основному класі, аби інші класи цієї можливості не мали.
 - b) Усі поля абстрактного класу GameEntity має рівень доступу protected. Це означає, що доступ до них мають лише ті класи, яка наслідують основний клас.
 - c) Рівень доступу public мають різні методи та властивості, що дозволяє їх використовувати у будь-якій частині коду.
4. Наслідування використовується для створення ієрархії класів
- a) Класи Tank і Bomb наслідують клас GameEntity, отримуючи базову функціональність, використовуючи її для себе.

Висновки

Отже, у третьому розділі ми надали відповідну програмну реалізацію коду, архітектуру якого ми надали у другому розділі. Ми трансформували всі наші класи колізій, об'єктів, та інших ігрових стуностей були успішно вкорінені у код, і візуально показали це. Надавши достатньо рисунків, ми показали, що створений ігровий застосунок дійсно є доволі комфортним та показує максимальну взаємодію з гравцями.

Також було показано, як саме принципи Ооп було реалізовано у програмі та які зв'язки між класами присутні.

ВИСНОВОК

Дана курсова робота була присвячена розробці ігрового застосунку "Tanks", що стало комплексним проєктом із застосуванням ключових принципів об'єктно-орієнтованого програмування та сучасних підходів до створення 2D-ігор із використанням бібліотеки SFML на мові C#. Метою роботи було створення повноцінного ігрового застосунку, який демонструє вміння використовувати концепції ООП та ефективні алгоритми для побудови ігрової динаміки. Всі поставлені завдання були успішно вирішені.

У процесі виконання роботи були досягнуті наступні ключові результати:

1. Теоретичне обґрунтування та застосування ООП:

- Вступ роботи сформував методологічну основу, визначивши ключові принципи ООП (поліморфізм, інкапсуляція, успадкування, абстракція) та їхнє практичне застосування в контексті проєкту.
- Надано чітке тлумачення цих концепцій, що демонструє глибоке розуміння парадигми ООП та її важливості для створення масштабованих та підтримуваних програмних систем.

2. Розробка алгоритмів ігрової динаміки та взаємодії:

- У першому розділі роботи було детально розглянуто принципи та будову взаємодії ігрових об'єктів.
- Застосовано алгоритм попиксельної перевірки зіткнень, який забезпечує високу точність, враховуючи реальну форму спрайтів, при цьому оптимізуючи обчислення за рахунок попередньої грубої перевірки прямокутниками для уникнення зайвих ітерацій.
- Реалізовано керування рухом та взаємодію об'єктів під час колізій за допомогою базової векторної математики, що

дозволило обчислити правильний вектор напрямку, забезпечити прямолінійний рух відносно структури об'єктів та побудувати детальний алгоритм відштовхування.

- Для створення ефекту нескінченного ігрового поля інтегровано функціонал обгортання екрану (screen wrapping), що забезпечує появу об'єктів на протилежному боці після виходу за межі поля.

3. Комплексне проєктування програмного модуля:

- У другому розділі було проведено всебічний аналіз вимог до застосунку, що дозволило чітко визначити необхідний функціонал та потреби користувачів для комфортної взаємодії.

- Розроблена UML-діаграма класів ефективно візуалізувала архітектуру програмного модуля, відобразивши зв'язки між класами (асоціації, успадкування, реалізації, залежності, агрегації, композиції), що підтвердило дотримання принципів об'єктно-орієнтованого проєктування.

- Застосування блок-схем детально проілюструвало динамічну поведінку ключових алгоритмів, таких як послідовність запуску турніру та ігрових сесій, а також логіку обробки руху ігрових об'єктів. Це забезпечило глибоке розуміння внутрішніх процесів програми.

4. Ефективна програмна реалізація:

- У третьому розділі була надана безпосередня програмна реалізація, де всі спроектовані компоненти та алгоритми були успішно втілені в код.

- Всі класи колізій, ігрових об'єктів та інших ігрових сутностей були інтегровані в код, що дозволило створити функціональний та візуально привабливий застосунок.

- Реалізований застосунок "Tanks" є комфортним у використанні та демонструє максимальну взаємодію з гравцями, що підтверджує успішне виконання всіх поставлених вимог та завдань.

Таким чином, виконана курсова робота демонструє повний цикл розробки програмного забезпечення, від аналізу вимог та архітектурного проектування до безпосередньої реалізації та впровадження складних ігрових механік. Вдале поєднання теоретичних знань з ООП та практичних навичок роботи з ігровими бібліотеками дозволило створити функціональний та якісний ігровий застосунок. Отримані результати підтверджують вміння автора застосовувати сучасні підходи до розробки програмного забезпечення та вирішувати комплексні інженерні завдання.

Список інформаційних джерел

1. Monogame documentation. [Електронний ресурс]. Режим доступу: https://docs.monogame.net/articles/getting_started/index.html. Перевірено 26.05.2025.
2. SFML documentation. [Електронний ресурс]. Режим доступу: <https://www.sfml-dev.org/documentation/3.0.1>. Перевірено 26.05.2025.
3. 2 3 4 Player Games/PlayMarket. [Електронний ресурс]. Режим доступу: <https://play.google.com/store/apps/details?id=com.ction.playergames&hl=en&pli=1>. Перевірено 26.05.2025.
4. tankGame. [Електронний ресурс]. Режим доступу: <https://github.com/yousefh409/tankGame/tree/master>. Перевірено 26.05.2025.
5. Основні принципи ООП. [Електронний ресурс]. Режим доступу: <https://campus.epam.ua/ua/blog/275>. Перевірено 23.05.2025.
6. Nick Walton. Pixel Perfect Collision Detection in C. [Електронний ресурс]. Режим доступу: <https://www.youtube.com/watch?v=9pnEBa4cy5w&t=130s>. Перевірено 24.05.2025.
7. UMLet. [Електронний ресурс]. Режим доступу: <https://umlet.com>. Перевірено 25.05.2025

Додатки

Додаток А. Лістинг коду алгоритму «Pixel perfect collision detection»

```
public static class PixelPerfectCollision
{
    public static byte[] CreateMask(Texture tx)
    {
        var img = tx.CopyToImage();
        int W = (int)tx.Size.X, H = (int)tx.Size.Y;
        var mask = new byte[W * H];
        for (int y = 0; y < H; y++)
            for (int x = 0; x < W; x++)
                mask[x + y * W] = img.GetPixel((uint)x, (uint)y).A;
        return mask;
    }

    public static bool Test(Sprite s1, byte[] mask1, Sprite s2, byte[] mask2,
byte alphaLimit = 0)
    {
        var r1 = s1.GetGlobalBounds();
        var r2 = s2.GetGlobalBounds();

        if (!r1.Intersects(r2, out FloatRect inter))
            return false;
        for (int yi = 0; yi < inter.Height; yi++)
        {
            for (int xi = 0; xi < inter.Width; xi++)
            {
                float wx = inter.Left + xi;
                float wy = inter.Top + yi;

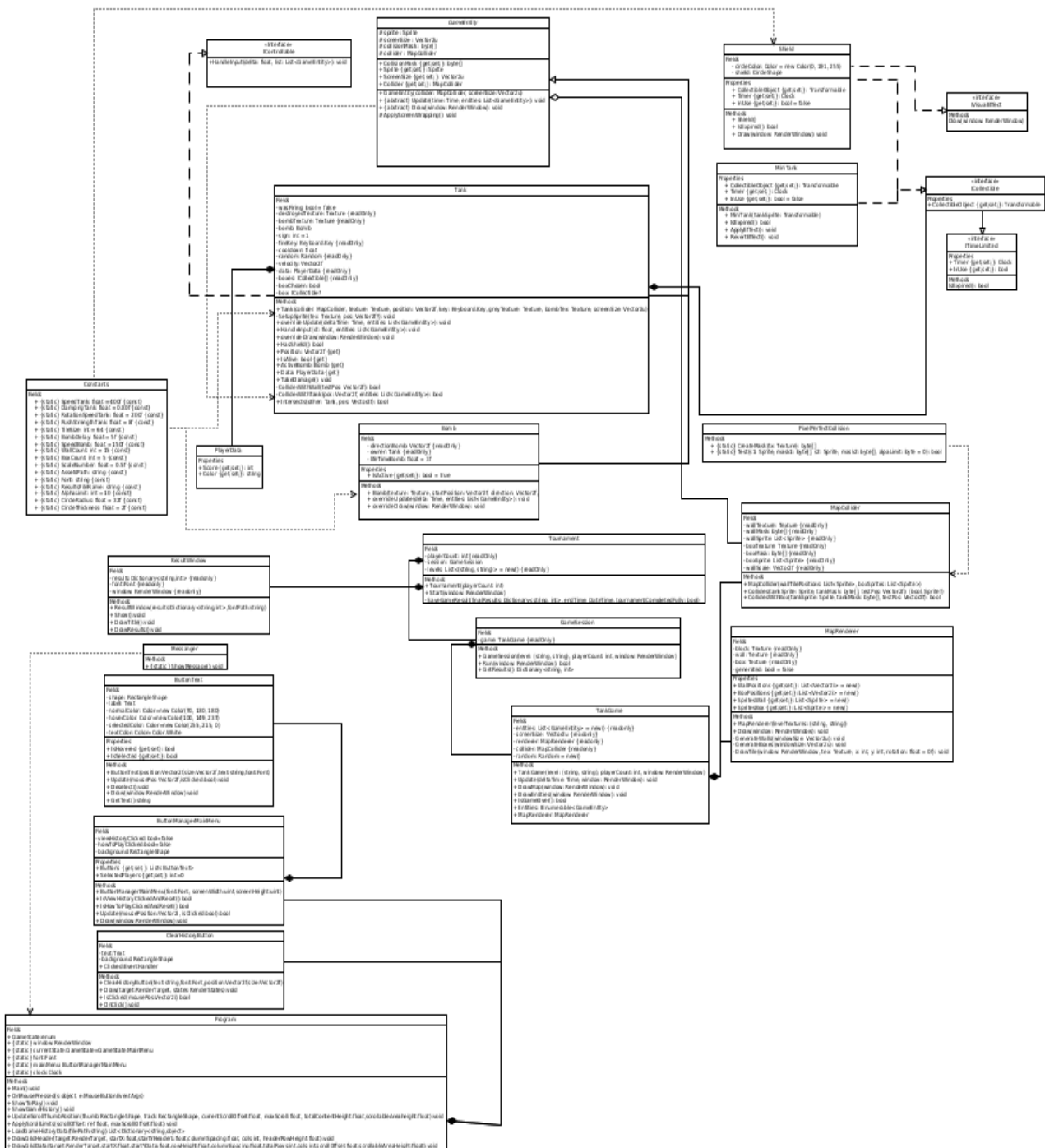
                var p1 = (Vector2f)s1.InverseTransform.TransformPoint(wx, wy);
                var p2 = (Vector2f)s2.InverseTransform.TransformPoint(wx, wy);

                int ix1 = (int)p1.X, iy1 = (int)p1.Y;
                int ix2 = (int)p2.X, iy2 = (int)p2.Y;

                if (ix1 >= 0 && iy1 >= 0 && ix2 >= 0 && iy2 >= 0 &&
                    ix1 < s1.TextureRect.Width && iy1 < s1.TextureRect.Height &&
                    ix2 < s2.TextureRect.Width && iy2 < s2.TextureRect.Height &&
                    mask1[ix1 + iy1 * s1.TextureRect.Width] > alphaLimit &&
                    mask2[ix2 + iy2 * s2.TextureRect.Width] > alphaLimit)
                    return true;
            }
        }

        return false;
    }
}
```

Додаток Б. UML-діаграма ігрового застосунку «Tanks».



Додаток В. Блок-схема роботи методу Start() класу Tournament

