Міністерство освіти і науки України
Черкаський національний університет імені Богдана Хмельницького
Факультет обислювальної техніки, 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, вч. 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)

Черкаси - 2025

ЗМІСТ

ВСТУП

Розробка ігор наразі є доволі динамічним та інноваційним напрямком у сучасній галузі розробки програмних застосунків. Створення будь-якого ігрового застосунку включає необхідність використання об’єктно-орієнтованого програмування аби задати правильну ієрархію класів, побудувати зв’язки між ними.

Метою даної роботи є створення ігрового застосунку, з використанням принципів ООП, демонстрації та застосування принципів створення 2D-ігор із засобами допоміжної бібліотеки SFML на мові програмування C#. Для цього поставлені наступні завдання.

1. Описати предметну область гри «2 3 4 Player Games».
2. Розробити алгоритми для побудови ігрового застосунку, руху об’єктів, взаємодії між ними.
3. Розробити 2D гру «2 3 4 Player Games» з використанням принципів ООП.
4. Протестувати розроблений застосунок «2 3 4 Player Games».

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 переваги та недоліки 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 фрейморк 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 аналізу 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 викокий контроль над кодом, пишучи власні методи використовуючи уже наявні, з використанням графіки тощо.

1.2 Порівняльний аналіз ігрових застосунків

Ігор, аби декілька гравців на одному екрані одночасно мали змогу грати у різноманітні міні гри, багато. Тому назва теми курсової роботи була взята уже з існуючого ігрового застосунку, – «2 3 4 Player games» [3]. Дана гра скаладається з багатьох міні ігор, які користувачі можуть вибрати і провести ігровий турнір для 2-4 гравців.

Було зроблено вибір саме створити декстопний варіант гри, так як вона сама є мобільною, з реалізацією однієї міні гри «Tanks». У ній користувачі керують своїми танками для знищення один одного.

Схожі ігри на тематику танків уже створювались користувачами, використовуючи SFML [4]. Даний застосунок також писався з використанням принципів ООП на мові C++.

1.3 Алгоритм взаємодії об'єктів у розроблюваному застосунку «2 3 4 Player Games»

У ігровому застосунку «2 3 4 Player Games» аби отримати правильні колізії з реальними формами об'єкта, було використано не просту перевірку меж спрайтів, у вигляді прямокутників (Bounding Box Collision), а попіксельну перевірку для спрайтів ігрових об'єктів та перешкод (Pixel Perfect Collision Detection) [5]. Під спрайтом у курсовій роботі розуміється ігровий об'єкт, перешкода та карта.

Алгоритм попіксельної перевірки спрайтів ігрових об’єктів 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 ігрових об’єктів 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, якщо прямокутники перетинаються, сам етап перевірки спрайтів об’єктів.

1.4 Векторна математика руху ігрових об’єктів у застосунку «2 3 4 Player Games»

Задля забезпечення правильної логіки руху ігрових об’єктів у застосунку, що розроблюється, 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 1.1.

$$\alpha = \alpha^{\circ} \cdot \frac{\pi}{180} \tag{1.1}$$

де α – кут у радіанах, α° – кут у градусах.

Після того, як обчислено кут, для обчислення вектора руху використовується формула 1.2.

$$u = (cos(\alpha), sin(\alpha)) \tag{1.2}$$

де u – вектор напрямку, $cos(\alpha)$ – X-складова вектору, $sin(\alpha)$ – Y-складова вектору.

Після цього множимо отриманий вектор на швидкість, з якою наш об’єкт буде рухатись у визначеному напрямку.

Тепер перейдемо до колізій. Коли ігровий об’єкт накладається на якийсь об’єкт (ігровий об’єкт чи перешкода), ми маємо заборонити його рух у цьому напрямку і відштовхнути його звідти. Отже, тут потрібно виконання обчислення та перевірити виконання умови відштовхування.

1. Обчислюється вектор різниці позицій (1.3):

$$D = P_1 - P_2 \tag{1.3}$$

де D – вектор різниці, P_1– позиція першого об’єкта, P_2– позиція першого об’єкта.

2. Обчислення довжини вектора різниці (1.4):

$$|D| = \sqrt{D_x^2 + D_y^2} \tag{1.4}$$

де $|D|$ – довжина вектор різниці, D_x– X-складова вектора різниці, D_y– Y-складова вектора різниці.

3. Перевіряється довжина вектора різниці на рівність нулю, якщо довжина нульова, знаходимо вектор відштовхування (1.5):

$$P_1' = P_1 + \frac{D}{|D|} S_{\text{відштовх}} \tag{1.5}$$

де P_1' – новий вектор першого об’єкта, $S_{\text{відштовх}}$– константа рівна *8f*.

Отже, саме з використанням цих формул вдається забезпечити правильне переміщення об’єктів у випадку визначеної колізії.

У застосунку буде використано обгортання екрану (Screen Wrapping) [7]. Цей алгоритм дозволяє об’єктам, які виходять за межі екрану, з’являтися на

протилежному боці екрану на тій же висоті. Для коректної обробки, потрібно перевіряти X та Y координати об’єкта на перетин з границями екрану (1.6 і 1.7)

$$P_x' = (P_x + W) mod W \tag{1.6}$$

$$P_y' = (P_y + H) mod H \tag{1.7}$$

де 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.

Висновки до першого розділу

У першому розділі було розглянуто передумови для розробки застосунку «2 3 4 Player Games». Було обрано платформу SFML для побудови ігрового застоснку, надали опис її та пояснили її вибір. Обґрунтувано тему курсової, адже вона була взята на основі уже існуючої мобільної гри. Надано інформацію про алгоритми та формули, що будуть використовуватись у проєкті.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ПРОГРАМНОГО МОДУЛЯ ДЛЯ ІГРОВОГО ЗАСТОСУНКУ «2 3 4 Player Games»

2.1 Аналіз вимог до ігрового застосунку «2 3 4 Player Games»

Перш ніж спроєктувати 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 гри у застосунку зі збереженням рівних прав користувачів. Тому для цього створено 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 кнопки взаємодії розміщені окремо для кожного гравця.
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 від того, чи перерваний був сам турнір, чи він був проведений до кінця.

Інтерфейс та навігація має надавати функції згідно заданих умов:

- головне меню з вибором функціоналу: кнопки вибору кількості гравців, перегляду ігрових сеансів та інформації як грати;
- після вибору кількості гравців запустити ігровий процес;
- вивести фінальні результати.

Надамо також словник термінів нашої програми аби не було непорозумінь у подальшому використанні (табл. 2.1):

Таблиця 2.1

Словник термінів

Назва ігрового об’єкту	Опис
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 зменшшення 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 «2 3 4 Player Games»

Зобразимо роботу гри. Почнімо з меню, коли користувач саме запускає гру. Зобразімо сецанірій блок-схеми, коли користувач уже вибрав відповідну кількість гравців. Вона зображена у додатку А.

Отже, після того як ми отримали кількість гравців у головному класі, запускається турні. 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 турнурі.

Після того як запустилась ігрова сесія, запускається сама гра з танками, що є ключовим в управлінні ігрового процесу, який 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 зобразимо також структурну схему програми (рис. 2.1).

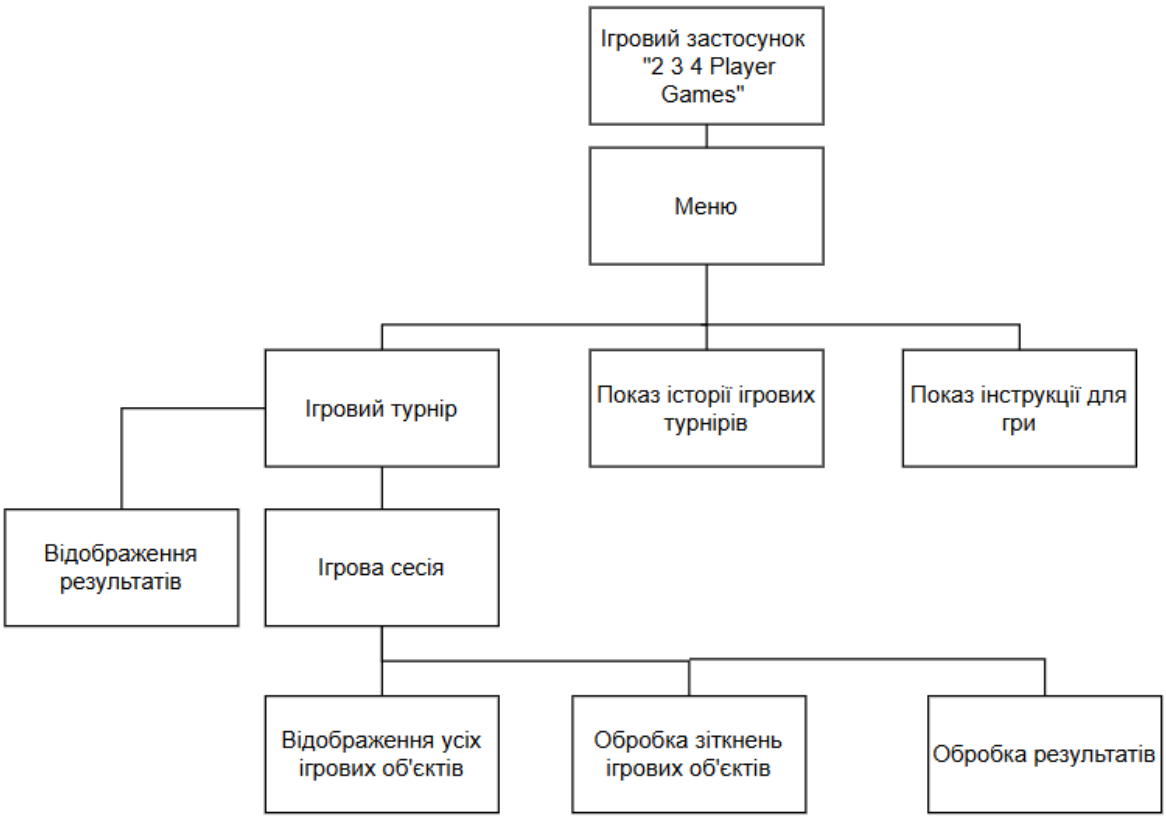


Рисунок 2.1. Структурна схема ігрового застосунку «2 3 4 Player Games»

Надана структурна схема представляє організацію та взаємодію компонентів ігрового застосунку "2 3 4 Player Games". Вона демонструє ієрархічну структуру, починаючи з головного застосунку і розгалужуючись на різні функціональні модулі та екрани.

Схема чітко показує модульну структуру програми. Вона розділена на логічні блоки, що відповідають за різні аспекти функціоналу: управління грою, відображення інформації, обробку ігрової логіки та взаємодію з користувачем (меню, інструкції, історія). Це типовий підхід до проектування програмного забезпечення, що сприяє легшому розробці, налагодженню та підтримці.

Висновки до другого розділу

Другий розділ був присвячений комплексному проєктуванню 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 другому було прописано блок-схеми та структурну схему ігрового застосунку «2 3 4 Player Games», що детально показують як працюють програмні компоненти: як проводяться турніри ігор, ігрові сесії, як рухаються об’єкти та інше.

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

3.1 Реалізація ігрової логіки в «2 3 4 Player Games»

Раніше у другому розділі були представлені алгоритми, що будуть використовуватись при реалізації застосунку «2 3 4 Player Games». Тепер детально проаналізуємо їх реалізацію у коді. У нас є основний клас TankGame, у якому створюються ігрові об’єкти Tank (у них передається колір танку), а також бомби, поточна позиція, відмальовується карта.

У класі Tank вміщуються основні методи, що відповідають за рух танків, їхнє зіткнення з ігровими об’єктами та відмалювання об’єктів на карті. Спочатку розберемо метод Update, код якого розміщений у додатку Б.

Основна частина методу постійно перевіряє, чи не зіткнувся об'єкт з "коробками-сюрпризами" (ігрові бонуси для гравця). При зіткненні, якщо ще не було обрано, випадковим чином обирається одна з доступних коробок (зменшення розмірів текстури танка або додавання щита) і активується її таймер. Коли час дії бонусу закінчується, його ефект скасовується до наступної активації.

Після обробки взаємодії з бонусами, метод вміщує в собі виклик іншого методу HandleInput, що обробляє введення користувача. Цей метод буде нижче описано. Інша частина методу вміщує фактичне переміщення танку на карті, що передбачає розрахунок бажаного кроку за поточний проміжок часу

За обробку руху танків та бомб через натискання клавішів відповідає метод HandleInput, код якого розміщено у додатку В. Метод HandleInput відповідає за обробку введення гравця, перевірки чи була натиснута клавіша, призначена для кожного танку, і після цієї перевірки здійснюється постріл чи рух танку.

Його робота починається з перевірки, чи танк не було знищено (!IsAlive). Якщо знищено – жодні дії з вводом не читаються, вихід з методу. Якщо ні, метод відстежує, чи натиснута клавіша (fireKey).

Логіка стрільби – якщо клавіша стрільби натиснута і час на відновлення стрільби вже минув, то виконуються наступні дії.

1. Розраховується напрямок польоту бомби на основі поточної повороту танка.
2. Створюється новий об'єкт Bomb (бомба) з відповідними параметрами.
3. Встановлюється таймер для стрільби, щоб запобігти повторному запуску бомби до кінця зазначеного часу.

Крім того, якщо клавіша просто натиснута, то незалежно від стрільби, танк починає свій рух вперед на основі свого повороту.

1. Розраховується напрямок руху.
2. До швидкості танка додається імпульс у цьому напрямку, помножений на швидкість танка та проміжок часу *dt*, що минув з останнього оновлення).

Як зазначалось раніше, рух танка та бомби розраховується відповідно позиції танка, коли він перестає обертатись. Танк обертається у випадку коли клавіша не натиснута.

1. Обертання танка обраховується за формулами 1.1 та 1.2 з можливістю зміни повороту: з права на ліво і навпаки.
2. Після спроби обертання виконуються дві важливі перевірки на зіткнення через клас PixelPerfectColision та його метод Test, розміщеного у додатку Г:
    - зіткнення з іншим танком;
    - зіткнення зі блоком-стіною.
3. Якщо виявляється зіткнення зіткнення присутнє, танк отримує відштовхувальний імпульс у позицію, в якій зіткнення з будь-яким об’єкто відсутнє (формули 1.3 – 1.5).

Як зазначалось, танк може міняти свій напрям обертання, чи то за годинникою стрілкою, чи проти. Якщо гравець щойно перестав стріляти (тобто wasFiring було true, а isFiringNow стало false), напрямок обертання змінюється на протилежний.

3.2 Реалізація графічного інтерфейсу в «2 3 4 Player Games»

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.

Для гри «2 3 4 Player Games» було реалізовано 5 кнопок (рис.3.1):

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.

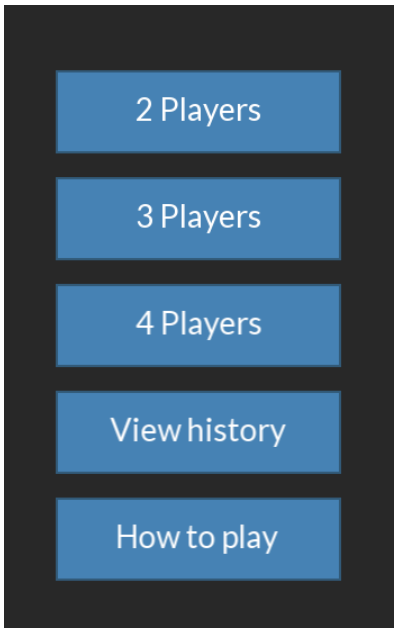


Рисунок 3.1. Кнопки основного меню гри «2 3 4 Player Games»

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 3.2).

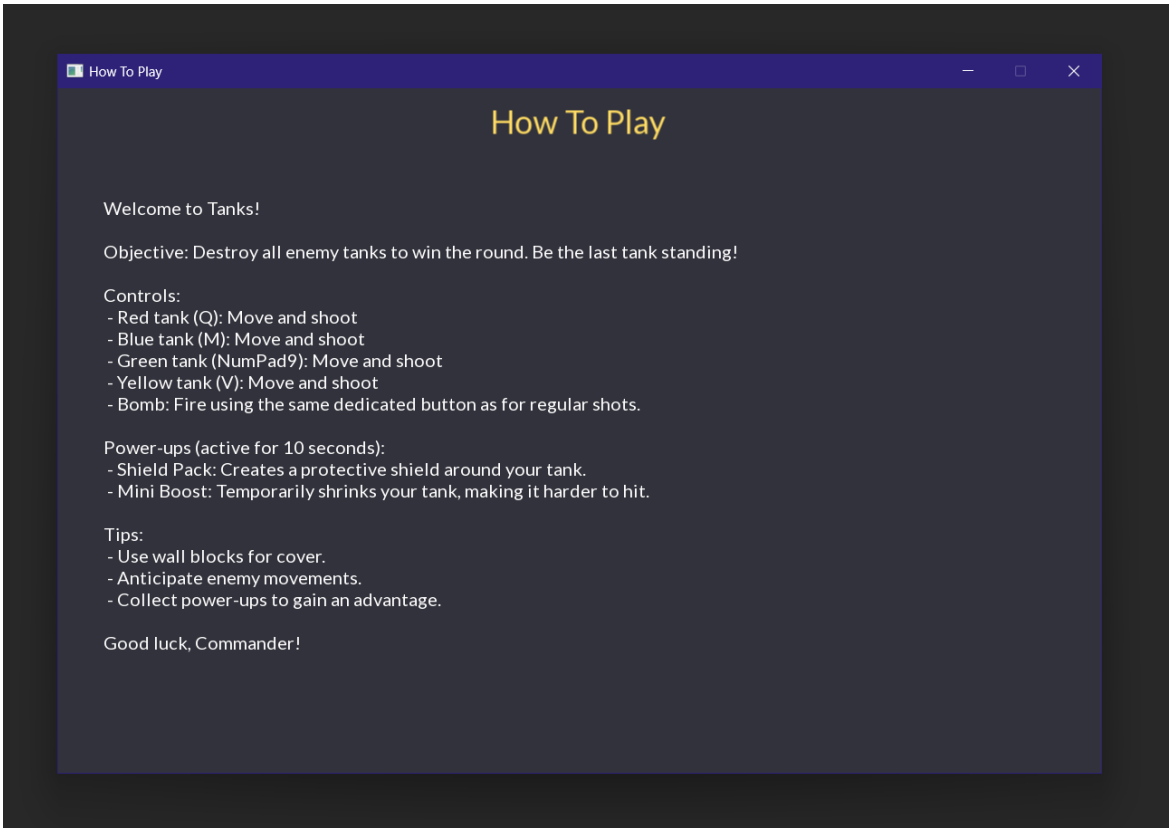


Рисунок 3.2. Вікно «How to play»

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

Тепер перейдемо до кнопки «View history» (рис 3.3.). 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 для очищення історії турнірів. Вона очищує відповідний файл, у якому зберігаються дані і опісля виводить вікно з текстом «History was removed!» (рис. 3.4.). Після цього історія буде очищена.

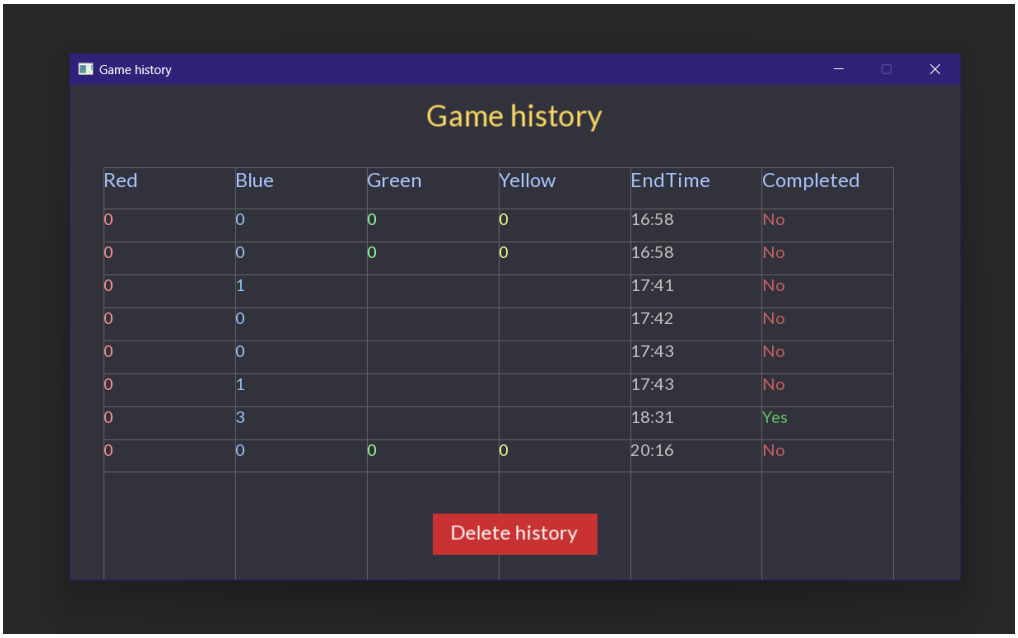


Рисунок 3.3. Вікно «Game history»

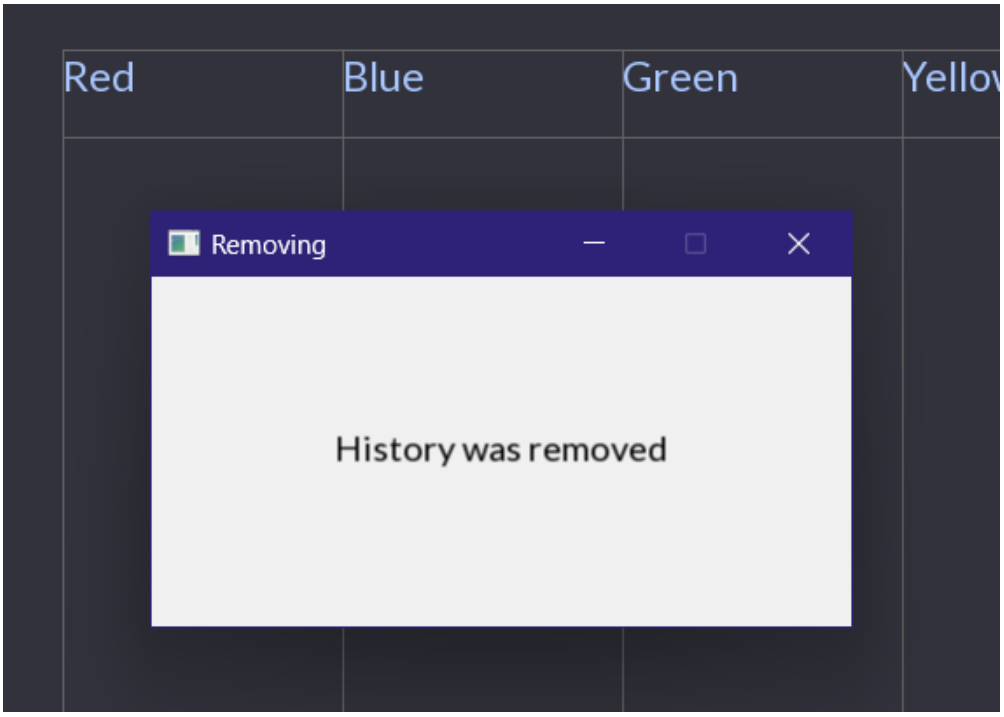


Рисунок 3.4. Вікно «Removing»

Далі йде функціонал кнопок з вибором гравців. У них він ідентичний, і запускає гру з відповідною кількостю гравців. На рисунку 3.5 показано ігровий турнір для 4 гравців.

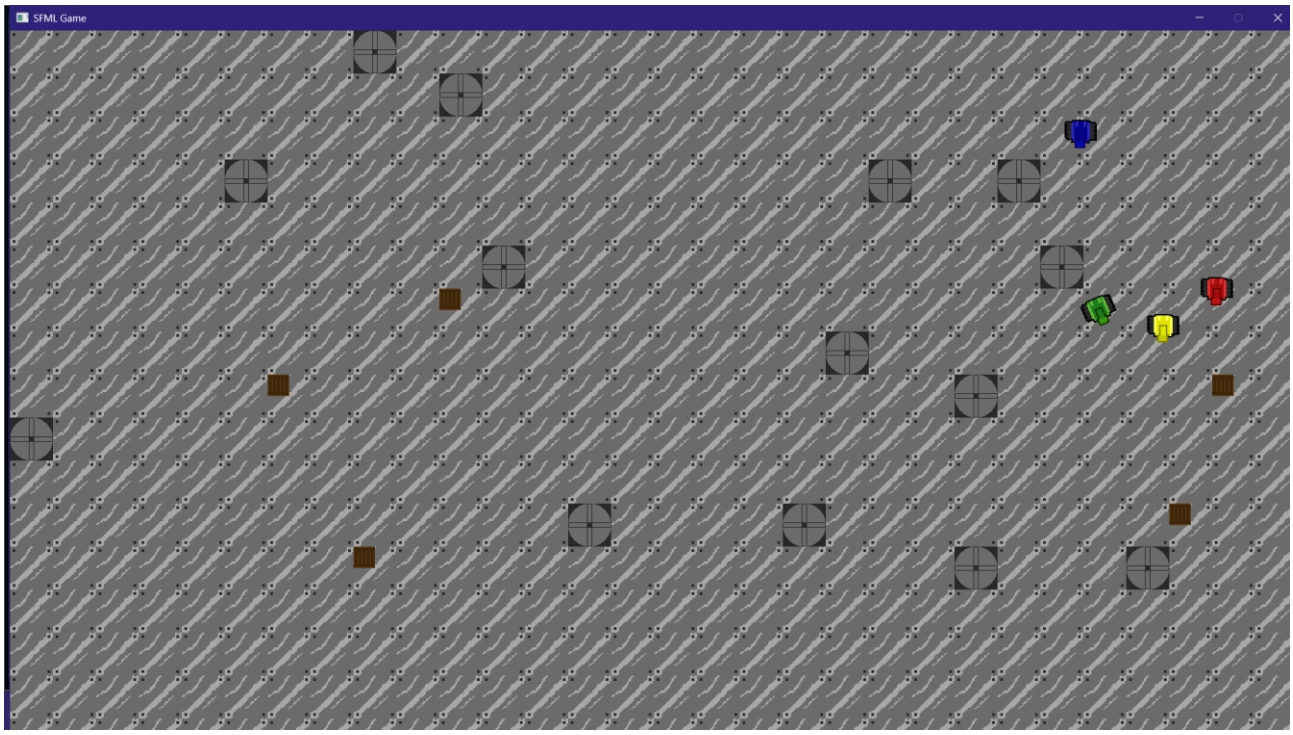
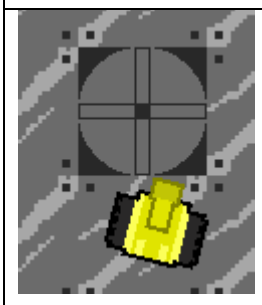
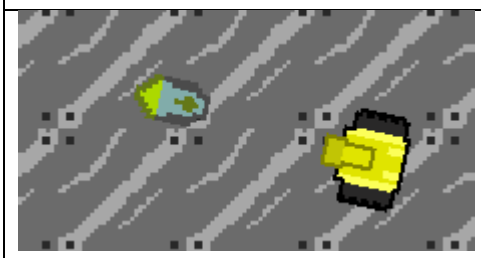


Рисунок 3.5.Ігровий турнір для 4 гравців

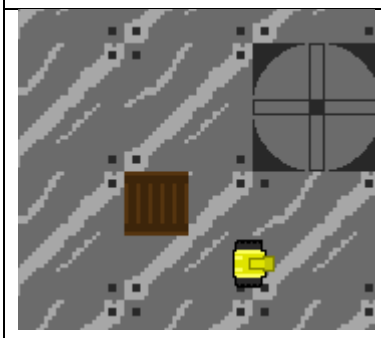
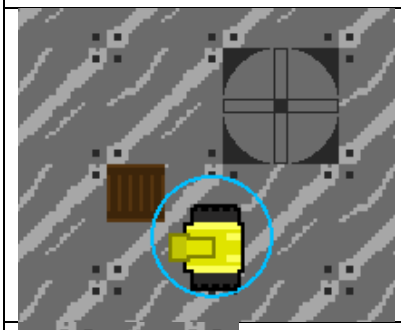
Тепер покажемо взаємодії всіх об’єктів на карті (табл. 3.1.)

Таблиця 3.1

Взаємодія об’єктів

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

Продовження таблиці 3.1

1	2
	Танк підібрав один з ящиків і у ньому зменшалась структура.
	Танк підібрав один з ящиків і у нього з’явився щит.
	Знищений танк після колізії з чужою бомбою

Саме такі взамодії присутні на карті. Задля досягнення правильних колізій було використано попередньо згаданий алгоритм попіксельної перевірки.

Тепер покажемо, які ще карти у нас присутні (рис 3.6 та рис.3.7).

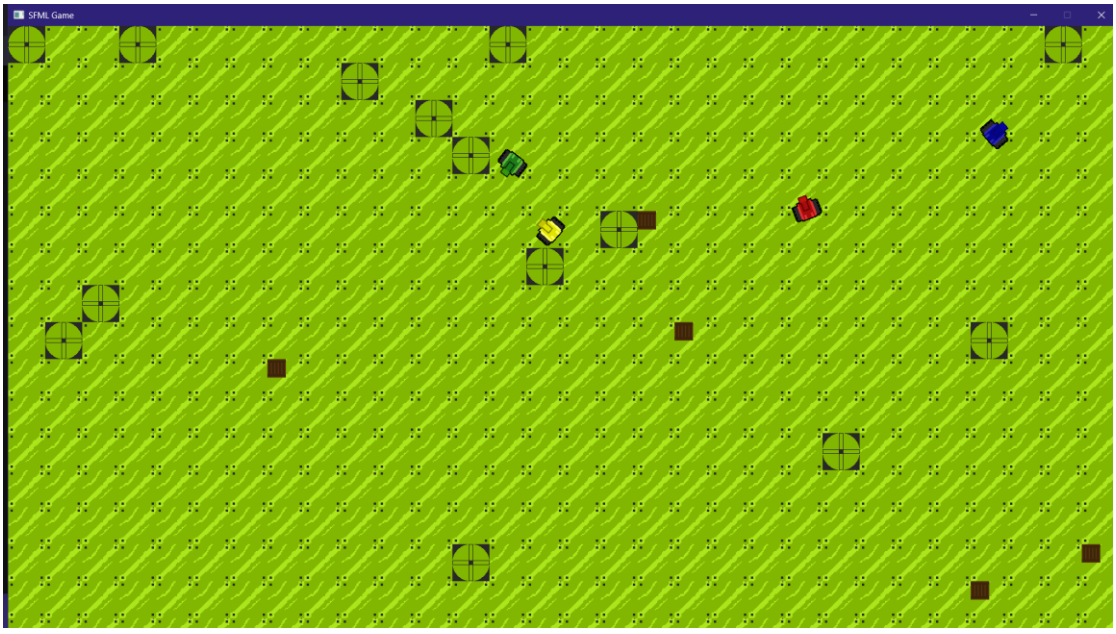


Рисунок 3.6. Друга ігрова сесія

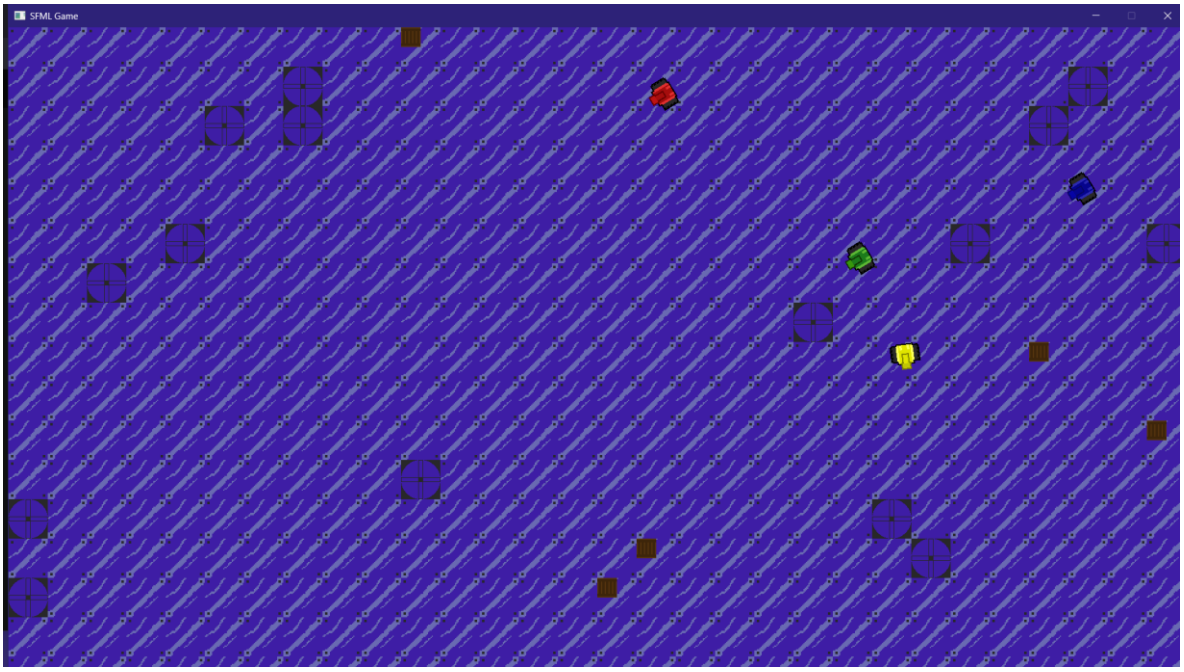


Рисунок 3.7. Третя ігрова сесія.

Після того як всі три ігрові сесії було проведено, відподно висвітлиться турнірна таблиця, в якій показано скільки в кожного гравця було набрано балів (рис 3.8.).

Рисунок 3.8. Турнірна таблиця

3.3 Реалізація ООП в ігровому застосунку «2 3 4 Player Games»

Перш за все наведемо UML-діаграму класів побудовану через додаток UMLet [6] . Вона розділена на два рисунки: на першому (додаток Д) розміщено класи, що пов’язані з логікою та структурою ігрового застосунку, на другому (додаток Е) показано як це пов’язується з класами, що формують меню та вивід результатів гри (обидва модулі пов’язуються через клас TankGame). Тепер розберемо детальніше як ООП реалізовується у програмі.

1. Абстракція дозволяє приховувати складні деталі реалізації, надаючи лише необхідний інтерфейс для взаємодії.
   a) Клас GameEntity є абстрактним, він визначає обов’язкові поля для ігрових об’єктів Tank і Bomb. Також має два абстрактних методи Update() та Draw(), деталі реалізацій яких залишаються за похідними класами.
   b) Клас Bomb() та Draw() використовують конструктор абстрактного класу, приймаючи значення полів MapCollider та Vector2u.
2. Поліморфізм дозволяє обробляти об’єкти різних класіу через єдиний інтерфейс.
   a) Як зазначалось раніше, клас GameEntity є абстрактним та має два абстрактних методи Update() та Draw(). Класи Tank і Bomb ці методи перевизначають і через поліморфізм ми маємо змогу створювати списки об’єктів GameEntity (наприклад такий список присутній у класі TankGame List<GameEntity> entities) і коли ігровий цикл перебиває ці списки, система автоматично визначає конкретний тип об’єкта і викликає відповідну реалізацію методу, що була перевизначена саме в цьому дочірньому класі. Тобто нам не потрібно створювати окремий екзмепляр вибраного класу.
   b) Також сюди відноситься імплементування інтерфейсів, що є у програмі. У класі Tank використовується масив ICollectible[] boxes, який містить Shield та MiniTank. Це демонструє поліморфізм: ми працюємо з різними типами об'єктів (Shield, MiniTank) через єдиний

інтерфейс Icollectible, здійснюємо такі перевірки як if (box is MiniTank miniTank) та if (box is IVisualEffect visualEffect && box.InUse). Це дозволяє нам, знову ж таки, викликати специфічні методи (ApplyEffect, RevertEffect, Draw) для цих об'єктів, використовуючи спільні інтерфейси.

3. Інкапсуляція активно використовується для приховування внутршньої 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 фінкціональність, використовуючи її для себе.

Висновки до третього розділу

Отже, у третьому розділі описано відповідну програмну реалізацію коду гри «2 3 4 Player Games», проєктування якого представлено другому розділі. Всі класи колізій, об’єктів, та інших ігрових стуностей були реалізовані у коді та знайшли відображення у ігорвому інтерфейсі. Також було показано, як саме принципи ООП допомогли реалізувати певні функції у застосунку.

ВИСНОВОК

Дана курсова робота була присвячена розробці ігрового застосунку «2 3 4 Player Games», що стало проєктом із застосуванням принципів об'єктно-орієнтованого програмування та підходів до створення 2D-ігор із використанням бібліотеки SFML на мові C#. У процесі виконання роботи були здійснено розробку алгоритмів ігрової динаміки та взаємодії, комплексне проєктування програмного модуля, ефективну програмну реалізацію.

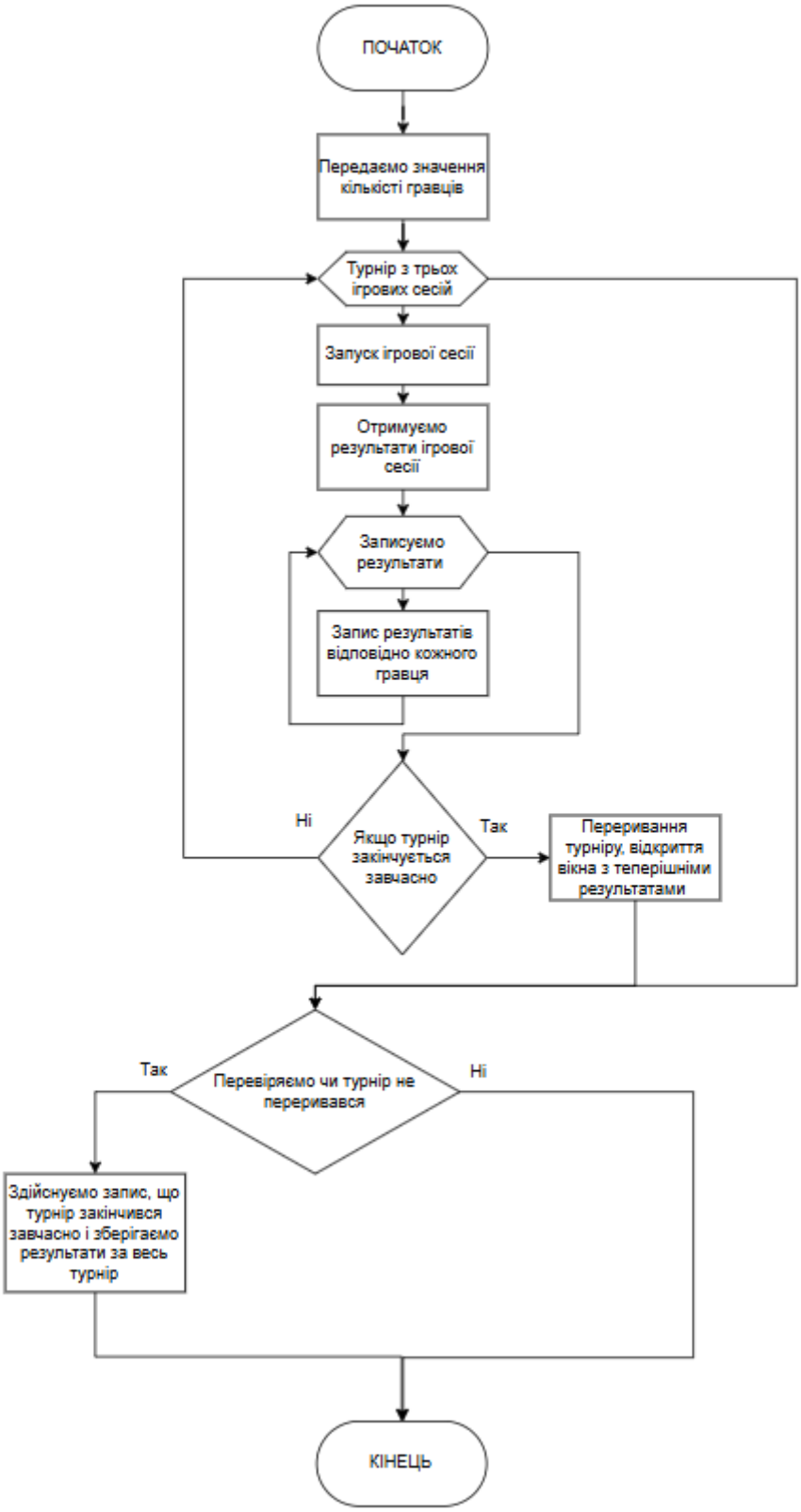
Таким чином, виконана курсова робота демонструє повний цикл розробки ігрового застосунку «2 3 4 Player Games», від аналізу вимог та проєктування до безпосередньої реалізації та впровадження складних ігрових механік. Вдале поєднання теоретичних знань з ООП та практичних навичок роботи з ігровими бібліотеками дозволило створити функціональний та якісний ігровий застосунок «2 3 4 Player Games».

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. 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.
6. UMLet. [Електронний ресурс]. Режим доступу: https://umlet.com. Перевірено 25.05.2025
7. Unity Basics – Screen Wrapping. [Електронний ресурс]. Режим доступу: https://www.youtube.com/watch?v=zWy29yeFNX8. Перевірено 05.06.2025

ДОДАТКИ

Додаток А. Блок-схема запусу турніру в ігровому модулі



Додаток Б. Реалізація методу Update() класу Tank

```
public override void Update(Time deltaTime, List<GameEntity> entities)
{
    float delta = deltaTime.AsSeconds();
    bomb?.Update(deltaTime, entities);
    if (cooldown > 0) cooldown -= delta;
    if (collider.CollidesWithBox(sprite, collisionMask, sprite.Position))
    {
        if (!boxChosen)
        {
            box = boxes[random.Next(boxes.Length)];
            boxChosen = true;
        }
        box?.Timer.Restart();
        box.InUse =true;
        if (box is MiniTank miniTank)
        {
            miniTank.ApplyEffect();
        }
    }
    if (box is not null && box.IsExpired())
    {
        box.InUse = false;
        if (box is MiniTank miniTank)
        {
            miniTank.RevertEffect();
        }
        boxChosen = false;
        box = null;
    }
    HandleInput(delta, entities);
    var step = velocity * delta;
    var pos = sprite.Position;
    var tryX = new Vector2f(pos.X - step.X, pos.Y);
    bool hitTankX = CollidesWithTank(tryX, entities);
    bool hitWallX = CollidesWithWall(tryX);
```

```
    if (!hitTankX && !hitWallX) pos.X = tryX.X; else velocity.X = 0;
    var tryY = new Vector2f(pos.X, pos.Y - step.Y);
    bool hitTankY = CollidesWithTank(tryY, entities);
    bool hitWallY = CollidesWithWall(tryY);
    if (!hitTankY && !hitWallY) pos.Y = tryY.Y; else velocity.Y = 0;
    sprite.Position = pos;
    velocity *= DampingTank;
    if (box is IVisualEffect visualEffect && box.InUse)
    {
        box.CollectibleObject.Position = sprite.Position;
    }
    ApplyScreenWrapping();
}
```

Додаток В. Реалізація методу HandleInput класу Tank

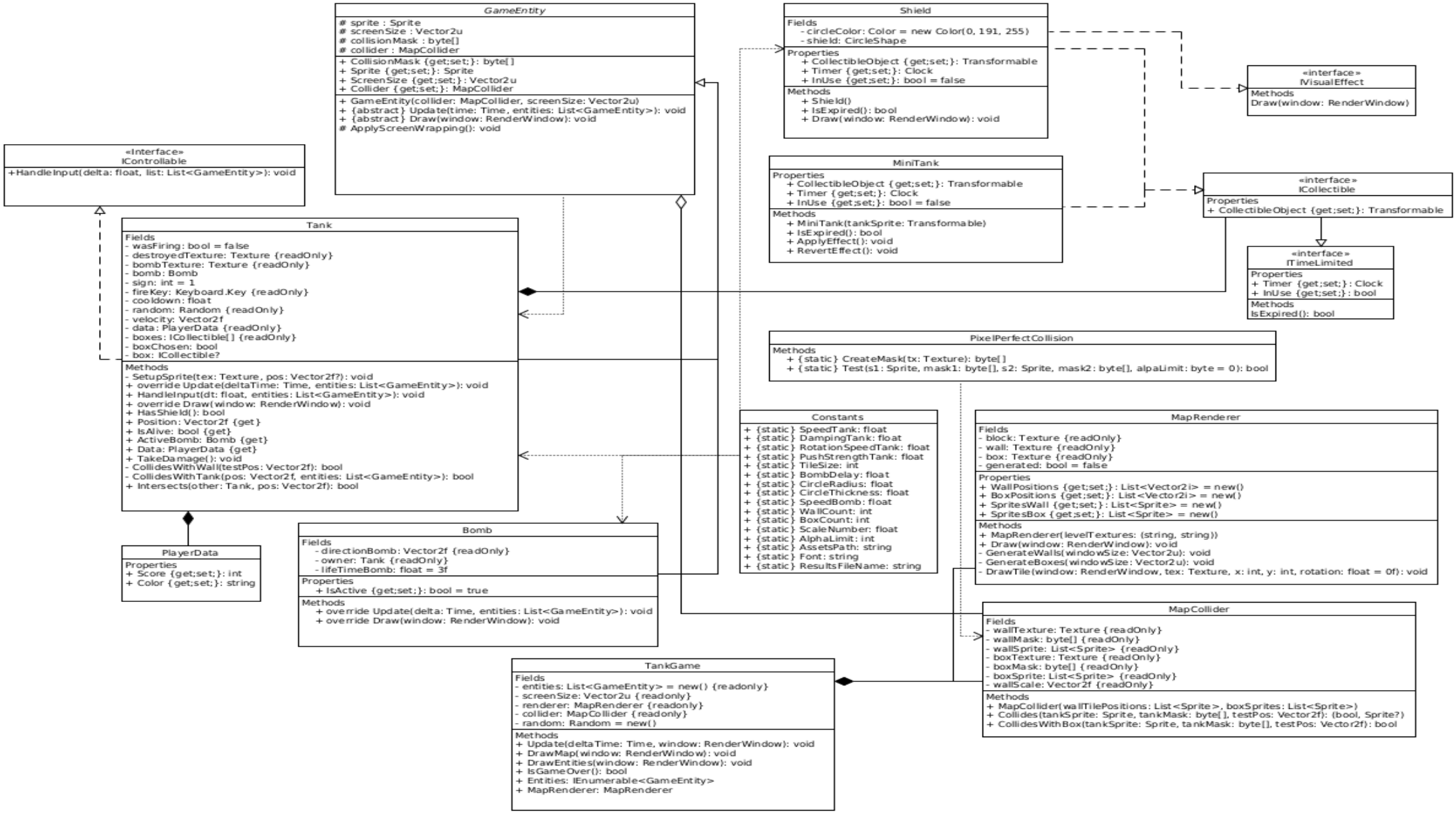
```
public void HandleInput(float dt, List<GameEntity> entities)
{
    if (!IsAlive) return;
    bool isFiringNow = Keyboard.IsKeyPressed(fireKey);
    if (isFiringNow && cooldown <= 0f)
    {
        float a = sprite.Rotation * (float)Math.PI / 180f;
        var dir = new Vector2f(-(float)Math.Sin(a), (float)Math.Cos(a));
        var spawn = sprite.Position + dir * 32f;
        bomb = new Bomb(bombTexture, spawn, dir, sprite.Rotation, this,
screenSize, collider);
        cooldown = BombDelay;
    }
    if (isFiringNow)
    {
        float a = (sprite.Rotation - 90f) * (float)Math.PI / 180f;
        var dir = new Vector2f((float)Math.Cos(a), (float)Math.Sin(a));
        velocity += dir * SpeedTank * dt;
    }
    else
    {
        float oldRot = sprite.Rotation;
        sprite.Rotation += sign * RotationSpeedTank * dt;
        var tankCollidesWIthOther = entities.OfType<Tank>()
                                .FirstOrDefault(t => t != this && Intersects(t,
sprite.Position));
        var positionWall = collider.Collides(this.sprite, this.collisionMask,
this.sprite.Position).Item2;
        if (tankCollidesWIthOther != null)
        {
            sprite.Rotation = oldRot;
            var diff = sprite.Position - tankCollidesWIthOther.Position;
            float len = (float)Math.Sqrt(diff.X * diff.X + diff.Y * diff.Y);
            if (len > 0) sprite.Position += (diff / len) * PushStrengthTank;
        }
```

```
        else if (positionWall is not null)
        {
            sprite.Rotation = oldRot;
            var diff = sprite.Position - positionWall.Position;
            float len = (float)Math.Sqrt(diff.X * diff.X + diff.Y * diff.Y);
            if (len > 0) sprite.Position += (diff / len) * PushStrengthTank;
        }
    }
    if (wasFiring && !isFiringNow)
    {
        sign = -sign;
    }
    wasFiring = isFiringNow;
}
```

Додаток Г. Фрагмент 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 Д. UML-діаграма класів модуля логіки ігрового застосунку «2 3 4 Player Games»



Додаток Е. UML-діаграма модуля меню та турніру ігрового застосунку «2 3 4 Player Games»