

Politechnika Śląska

Wydział Automatyki, Elektroniki i Informatyki

Programowanie Komputerów 4

Gra typu Tower Defence

Autor	Karol Ziaja
Prowadzący	Dr hab. inż. Roman Starosolski
Rok akademicki	2023/2024
Kierunek	Informatyka
Rodzaj studiów	SSI
Semestr	4
Termin zajęć laboratoryjnych	Wtorek, 14:15 – 15:45
Sekcja	12
Termin oddania sprawozdania	23-06-2024

1. Treść zadania

Gra jest inspirowana kultową serią gier Tower Defence – „Plants vs Zombies”.

W trakcie rozgrywki gracz musi bronić swojego domu przed kolejnymi, nadchodzącymi falami przeciwników. W tym celu może rozmieszczać różne rodzaje sojuszników na polu bitwy. Każda z jednostek ma swoje określone unikatowe zastosowanie: niektóre z nich są stricte ofensywne, niektóre służą do wzmocnienia obrony, a jeszcze inne do generowania zasobów dostępnych jako waluta w grze do zakupu jednostek. Istnieje również kilka rodzajów przeciwników. Różnią się oni od siebie ilością zdrowia oraz prędkością.

Każdy poziom składa się z kilku głównych fal przeciwników, a po przejściu danego poziomu gracz odblokowuje kolejne jednostki dostępne w grze oraz może przejść do kolejnego etapu, trudniejszego od poprzedniego. Ten sam poziom jest schematyczny pod względem trudności fal przeciwników, ale ich rozmieszczanie na planszy jest pseudolosowe i różne dla każdego podejścia. Oprócz głównego trybu gry dostępny jest również nieskończony tryb gry „survival”, w którym narastająca trudność rozgrywki poprowadzi do nieuchronnej porażki gracza, a jego wynik punktowy pojawi się w rankingu.

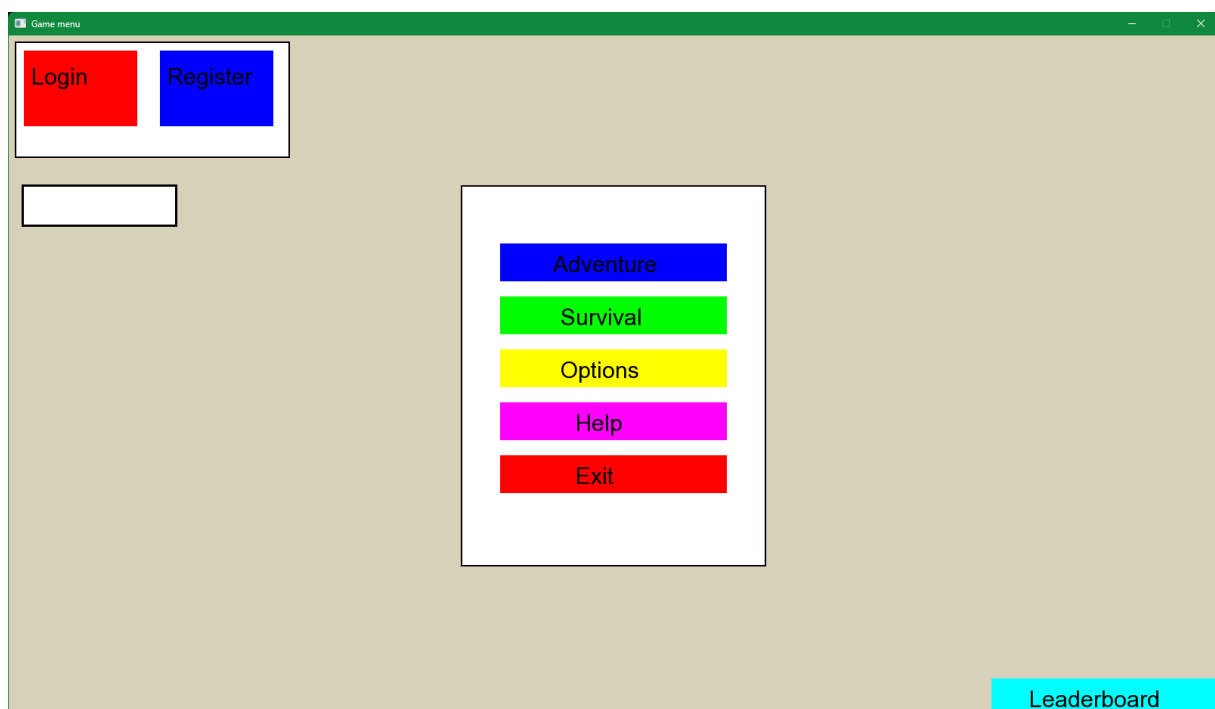
2. Analiza zadania

Projekt został napisany w języku C++ z wykorzystaniem zewnętrznej biblioteki SFML. Biblioteka w zamyśle służy do tworzenia prostych programów okienkowych lub rysowania kształtów, obsługuje akcelerację sprzętową grafiki 2D przy użyciu OpenGL. Biblioteka została wybrana ze względu na jej dobrą znajomość przez autora projektu i jego wcześniejsze projekty programistyczne. Wykorzystywane klasy, struktury danych i algorytmy zostały opisane w sekcji **Specyfikacja wewnętrzna**.

3. Specyfikacja zewnętrzna

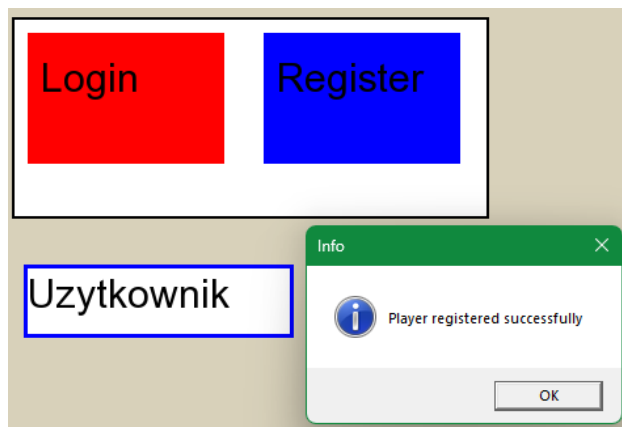
Po uruchomieniu programu użytkownik znajduje się w menu głównym. Bezpośrednio z menu gracz może wykonać kilka czynności:

- Rozpocząć nową grę w trybie **Adventure** lub **Survival**
- Przejść do okna z ustawieniami gry – **Options**
- Wyświetlić pomoc(instrukcję) – **Help**
- Wyjść z gry – **Exit**
- Wyświetlić tabelę z rankingiem – **Leaderboard**
- Zalogować się – **Login**
- Zarejestrować się – **Register**

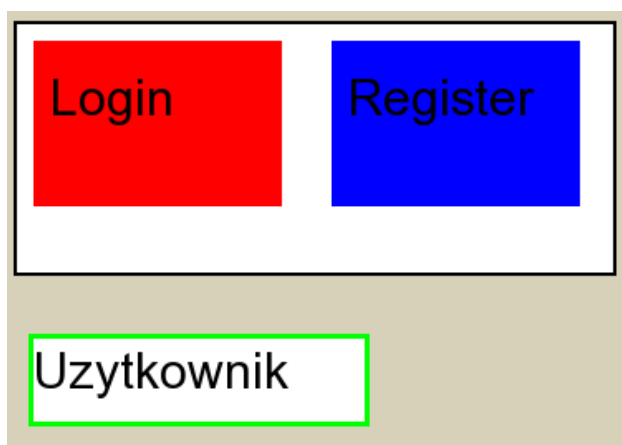


Zrzut ekranu 1: Menu główne programu

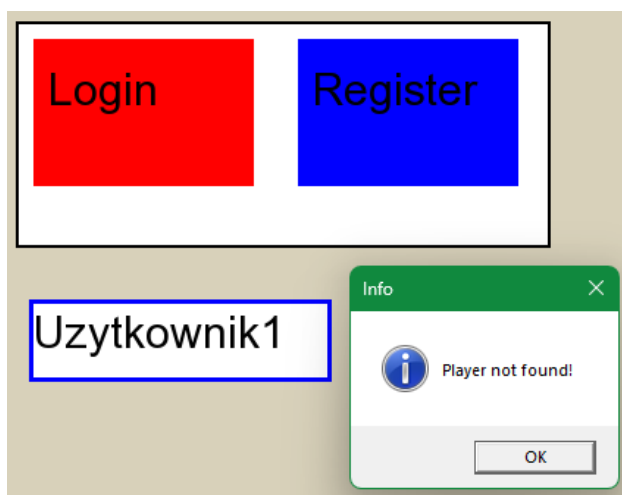
Zarówno rejestracja jak i logowanie korzysta z umieszczonego poniżej pola tekstowego, w którym należy wpisać nazwę gracza, a następnie kliknąć jeden z przycisków. Nazwa użytkownika powinna zaczynać się z wielkiej litery i składać się tylko i wyłącznie z liter.



Zrzut ekranu 2: Rejestracja nowego użytkownika

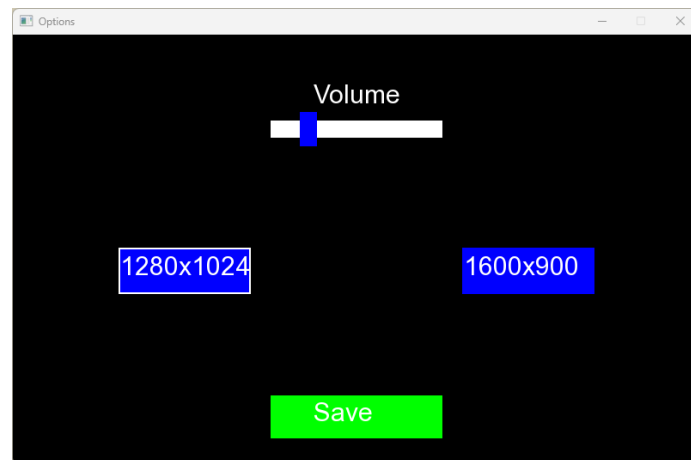


Zrzut ekranu 3: Pomyślne logowanie



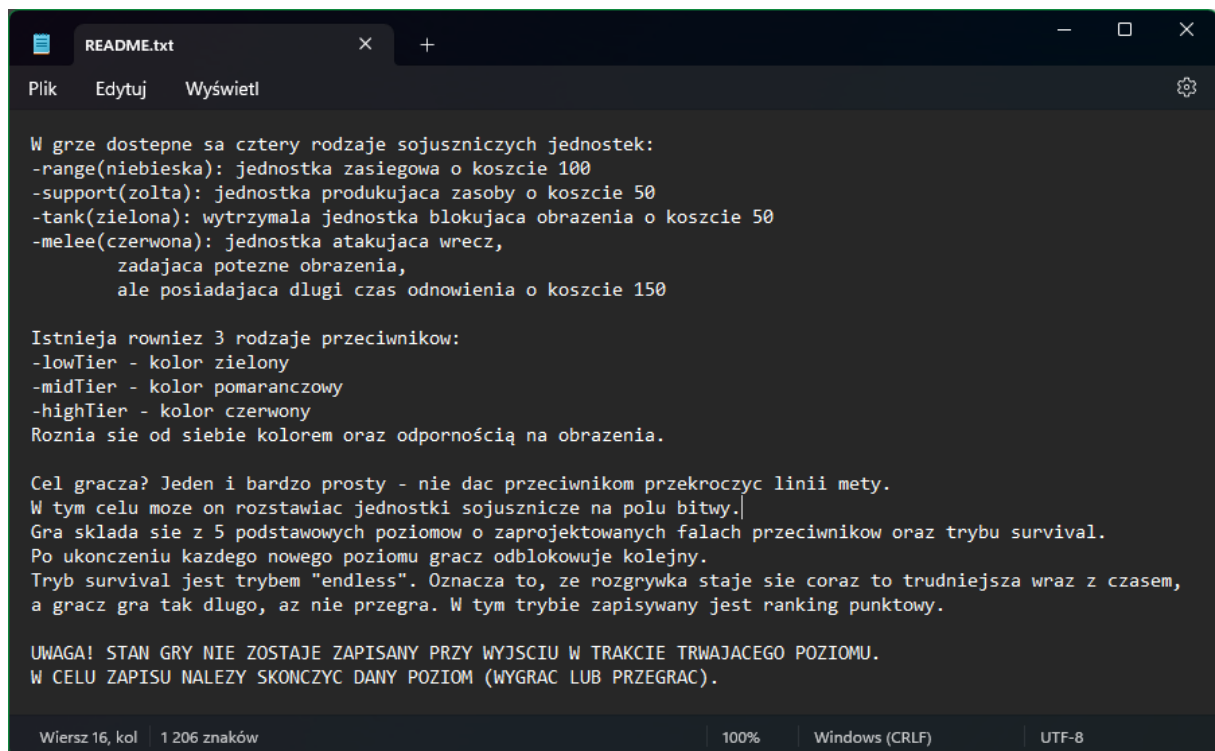
Zrzut ekranu 4: Logowanie zakończone niepowodzeniem

W okienku z ustawieniami rozgrywki użytkownik może zmienić rozdzielczość w jakiej wygeneruje się okno gry, a także zmienić poziom głośności za pomocą suwaka. Przycisk **Save** zapisuje ustawienia i wychodzi z tego okna do menu głównego.



Zrzut ekranu 5: Okno ustawień rozgrywki

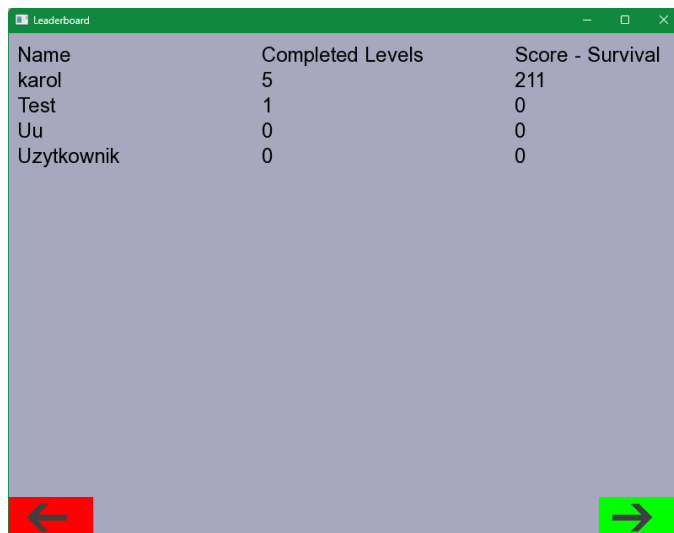
Kliknięcie przycisku **Help** w menu głównym skutkuje otwarciem pliku **README.txt**. Jest to plik zawierający instrukcję dla gracza. Jest w nim opisane oznaczenie każdego z sojuszników, koszty ich rozstawienia, typy przeciwników oraz główny zamysł rozgrywki. Poniżej zamieszczono zrzut ekranu zawierający te informacje.



Zrzut ekranu 6: Plik z instrukcją dla gracza

Kliknięcie w przycisk **Leaderboard** skutkuje otwarciem okna z rankingiem graczy.

Są tu zapisane takie informacje jak: nazwa użytkownika, liczba poziomów które ukończył w trybie Adventure, a także rekordowy wynik punktowy w trybie Survival. Na jednej stronie rankingu zmieścić się może maksymalnie 10 graczy, ale okno posiada przyciski oznaczone strzałkami. Służą one do przełączania pomiędzy kolejnymi stronami rankingu (jeśli takie istnieją).



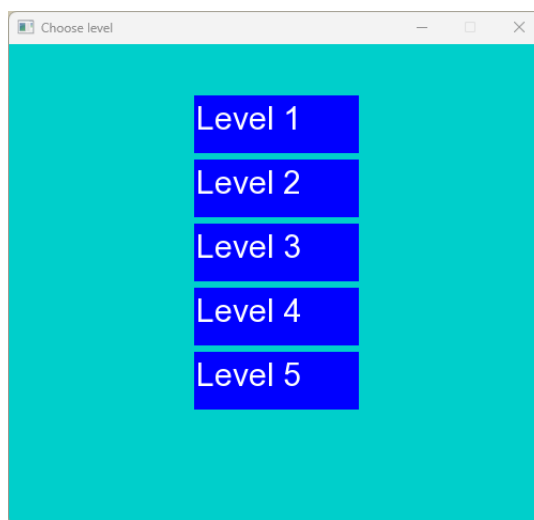
The screenshot shows a window titled "Leaderboard" with a table of player statistics. The table has three columns: "Name", "Completed Levels", and "Score - Survival". The data rows are as follows:

Name	Completed Levels	Score - Survival
karol	5	211
Test	1	0
Uu	0	0
Uzytkownik	0	0

At the bottom of the window, there are two buttons: a red button with a left arrow and a green button with a right arrow.

Zrzut ekranu 7: Tabela rankingowa

Jeśli gracz jest zalogowany, a ustawienia zostały skonfigurowane, możliwe jest przejście do rozgrywki. Kliknięcie w przycisk **Adventure** uruchamia okno wyboru poziomu.

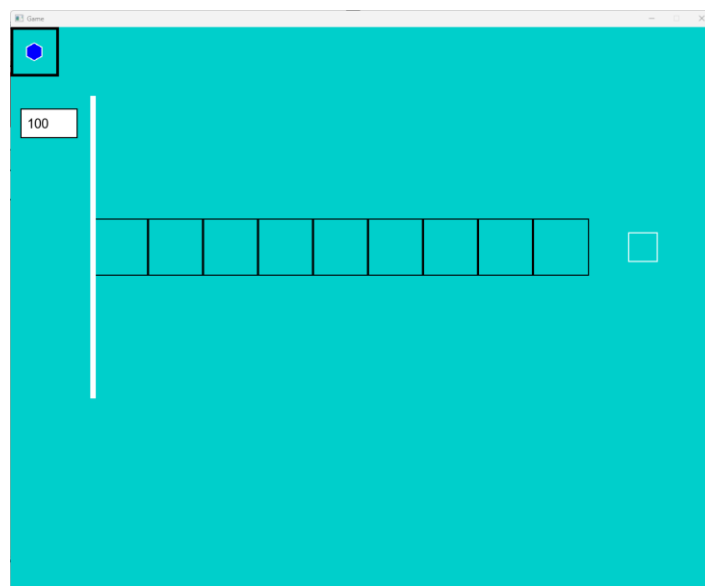


Zrzut ekranu 8: Menu wyboru poziomu

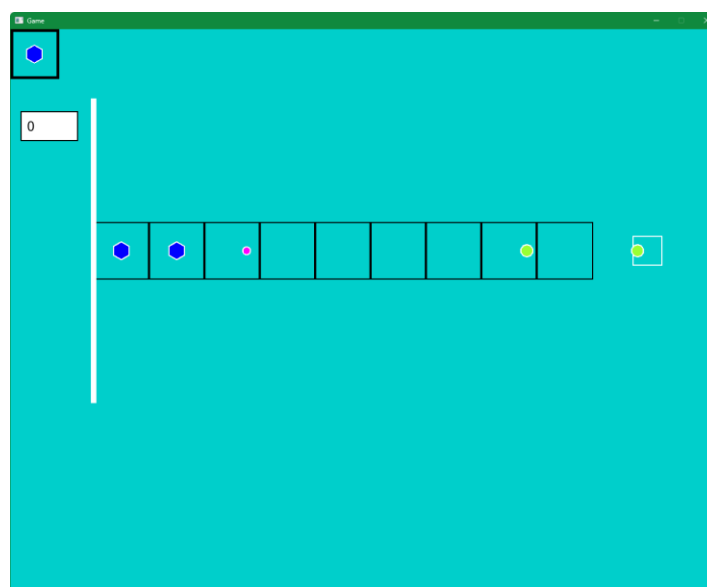
Każdy z poziomów różni się od siebie ilością dostępnych jednostek, falami przeciwników lub wielością pola bitwy. W skład tego okna wchodzi „dom” gracza – biała linia, plansza rozgrywki (czarne kwadraty), sojusznicze karty (w lewym górnym rogu) oraz ilość zasobów, umieszczona poniżej kart. Po prawej stronie planszy widoczny jest „spawner”, czyli miejsce pojawiania się przeciwników. Gracz może rozmieszczać jednostki po kliknięciu na jedną z kart. Ponowne kliknięcie na jakiegokolwiek pole rozgrywki umieści danego sojusznika pod dwoma warunkami:

- Gracz posiada wystarczającą liczbę zasobów
- Wybrane pole nie jest zajęte przez inną jednostkę

Poniżej umieszczono zrzuty ekranowe ilustrujące rozgrywkę w różnych momentach.

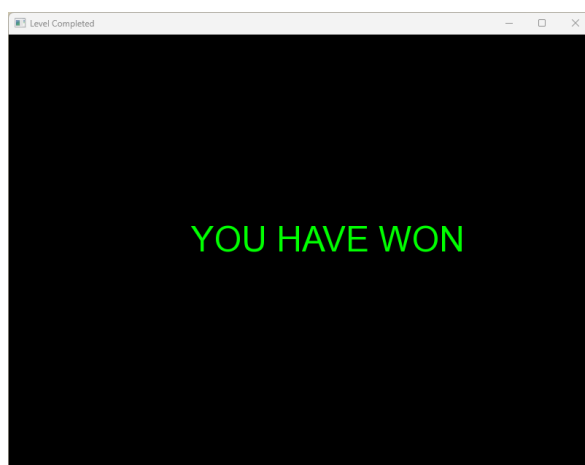


Zrzut ekranu 9: Poziom 1 – początek rozgrywki



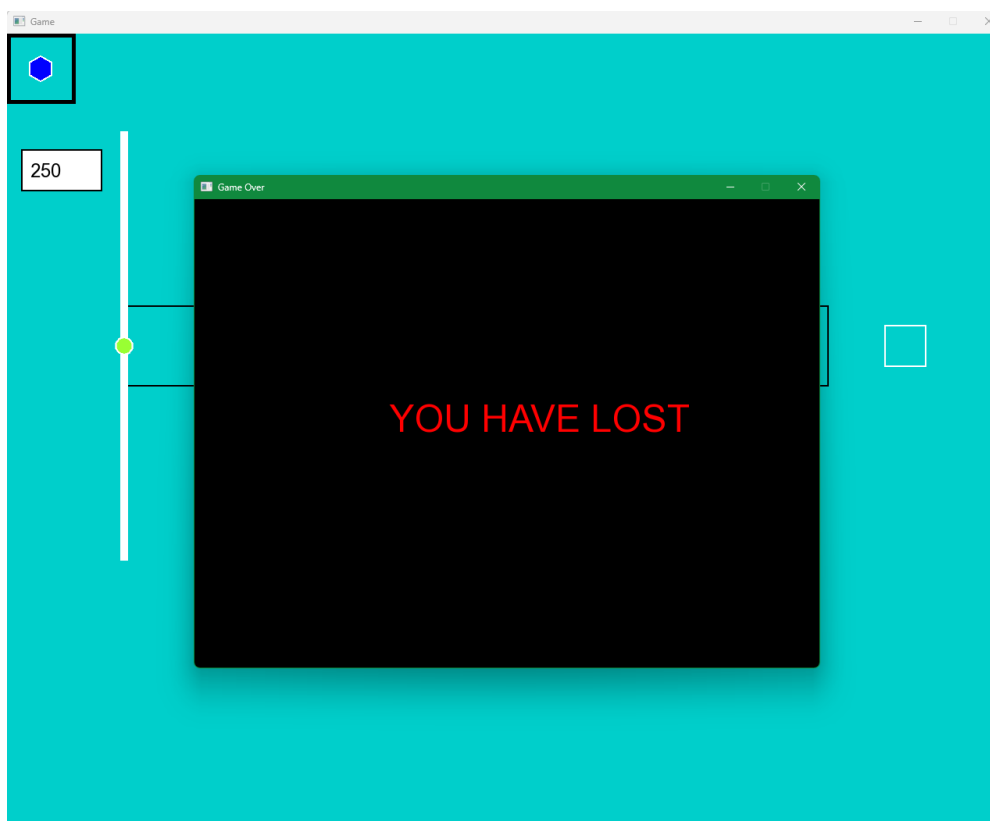
Zrzut ekranu 10: Poziom 1 – środek rozgrywki

Poniżej widoczne są ekrany zwycięstwa i porażki jakiegokolwiek poziomu. Zwycięstwo następuje, gdy pokonamy wszystkie fale przeciwników zanim Ci przekroczą białą linię. Jeśli zaliczony został najnowszy poziom dla gracza (gracz nie odgrywa ponownie starych poziomów, ale zaczyna nowy), następuje odblokowanie kolejnego.



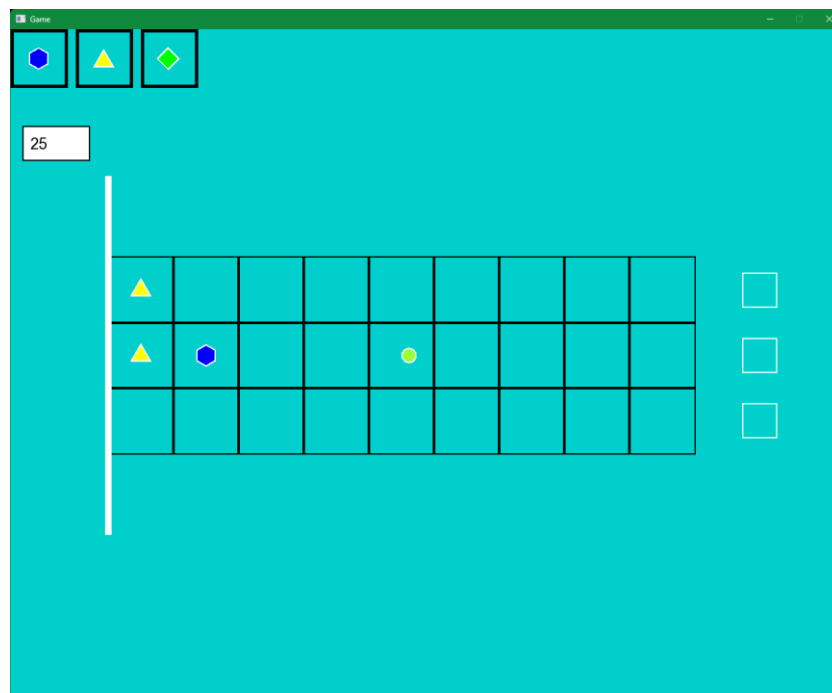
Zrzut ekranu 11: Ukończenie poziomu 1

Przegrana następuje, gdy którykolwiek z przeciwników przekroczy białą linię. W tym momencie pętla gry się kończy, a gracz widzi ekran porażki. Po jego zamknięciu oba okna zamykają się, a gracz przechodzi do menu głównego.

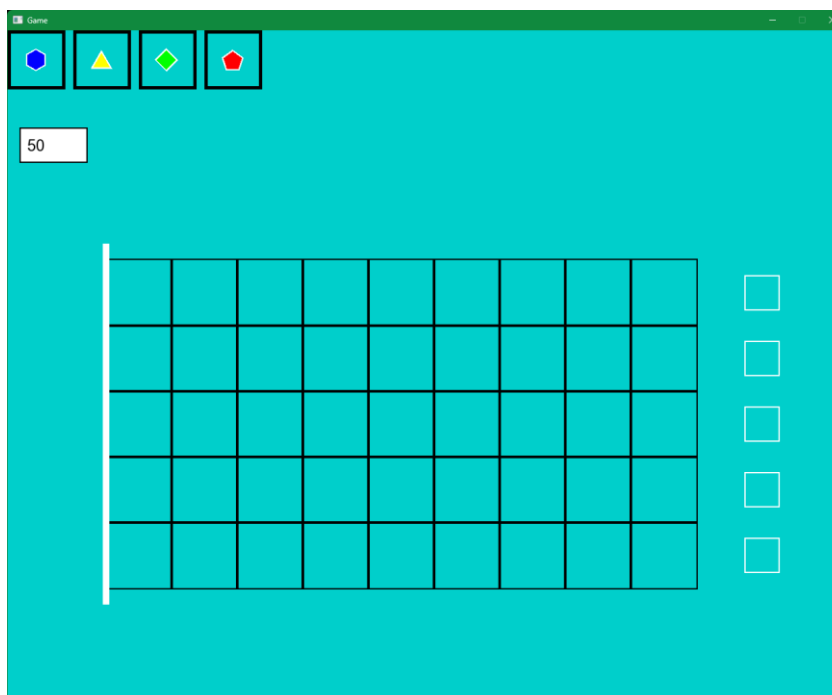


Zrzut ekranu 12: Przegrana poziomu 1

Poniżej przedstawiono przykładowy etap rozgrywki na poziomie 3 oraz 5. Możemy zauważyć dodatkowe karty jednostek oraz powiększenie się planszy gry.



Zrzut ekranu 13: Rozgrywka na poziomie 3



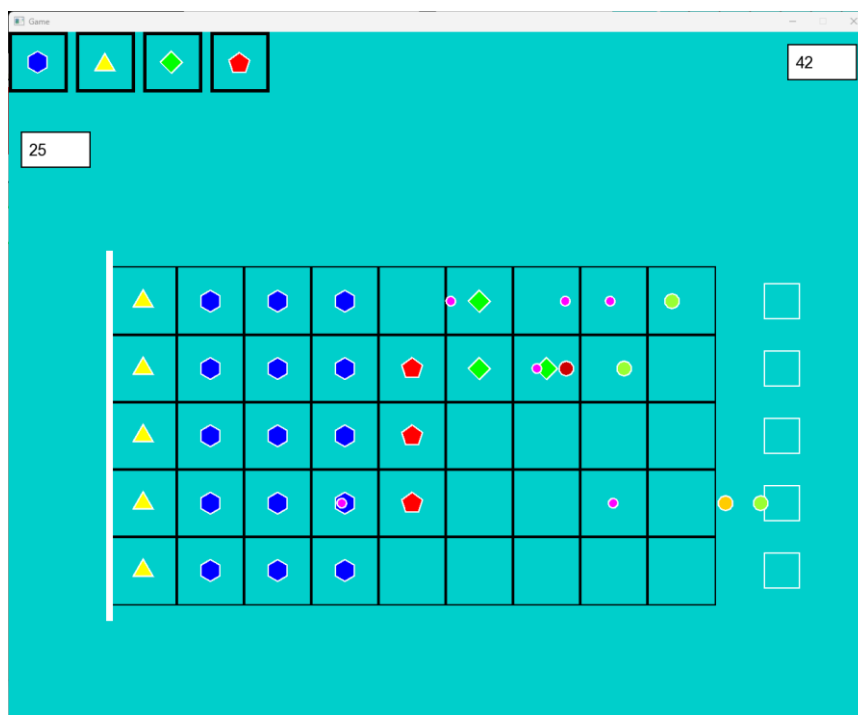
Zrzut ekranu 14: Początek rozgrywki na poziomie 5

W menu głównym pozostaje ostatni przycisk, czyli **Survival**. Kliknięcie go uruchomi pętlę gry, lecz z innym generowaniem fali przeciwników. W tym trybie gracz zaczyna ze zwiększoną liczbą zasobów, aby mógł utworzyć wstępną linię obrony. Początkowo przeciwnicy pojawiają się co 15 sekund. Jednakże czas ten maleje o 5% z każdym kolejnym pojawieniem się przeciwnika, aż do minimum – 0,5 sekundy. Wprowadzono również dodatkowe utrudnienie:

- co 5 przeciwnik jest typu midTierEnemy
- co 25 przeciwnik jest typu highTierEnemy

Ten tryb zawiera dodatkowo okienko reprezentujące aktualny wynik punktowy, znajdujące się w prawym górnym rogu.

Gra w tym trybie dąży do nieuchronnej porażki gracza w taki sam sposób jak w poziomach Adventure, ale ranking punktowy jest zapisywany (o ile osiągnięto nowy rekord).



Zrzut ekranu 15: Rozgrywka w trybie survival

4. Specyfikacja wewnętrzna

4.1. Struktury danych

Program przechowuje większość danych w wektorach:

- `std::vector<allyCard> allyCards` – wektor przechowujący karty postaci sojusznicznych `allyCard`. Jego rozmiar definiuje ilość kart, a więc ilość różnych typów sojuszników które możemy rozmieszczać na danym poziomie.
- `std::vector<std::vector<slot>> slots` – wektor przechowujący 2-wymiarową planszę gry składającą się z obiektów typu `slot`.
- `std::vector<spawner> spawners` – wektor przechowujący obiekty typu `spawner` służące do graficznej reprezentacji miejsc pojawiania się przeciwników. Jego wielkość zależy od ilości „wierszy” na planszy gry, czyli od pierwszego wymiaru wektora `slots`.
- `std::vector<missile*> missiles` – wektor przechowujący wskaźniki na wszystkie pociski `missile` wystrzelone przez jednostki zasięgowe. Pojemność wektora zwiększa się w momencie oddania strzału i zmniejsza, gdy nastąpi kolizja obiektu z przeciwnikiem lub oknem gry. Elementy tego wektora są tworzone dynamicznie.
- `std::vector<enemy*> enemies` – wektor przechowujący wskaźniki na wszystkich przeciwników pojawiających się w grze. Podobnie jak wyżej, przeciwnicy są tworzeni dynamicznie. Klasa `enemy` jest abstrakcyjna, co znaczy, że wektor przechowuje wskaźniki na obiekty klas pochodnych po `enemy`.

Oprócz wektorów program wykorzystuje statyczną zmienną klasy `TextureManager` w celu przechowywania wczytanych tekstur w mapie:

- `static std::map<std::string, sf::Texture*> textures`

Odpowiednie metody klasy pozwalają na dostęp do wczytanych tekstur.

4.2. Algorytmy

Podstawowym algorytmem działania programu jest cała pętla gry. To w niej zaimplementowane są wszystkie kolizje, wywoływanie metod akcji jednostek, obsługa zdarzeń oraz warunki końca rozgrywki.

Program wykorzystuje jeden klasyczny algorytm jakim jest **sortowanie**.

```
std::ranges::sort(enemies, [](enemy* a, enemy* b) {  
    return a->getXPosition() < b->getXPosition();  
});
```

Powyższy kod jest częścią metody szukającej najbliższego przeciwnika dla sojuszniczej jednostki. Jednym z kroków jest właśnie posortowanie wektora przeciwników ze względu na ich pozycję. Posortowany wektor następnie przechodzi przez różne warunki filtracji.

4.3.Opis klas

- Ally – klasa abstrakcyjna reprezentująca jednostki sojusznicze. Klasa dziedziczy po sf::Drawable, więc obiekty mogą być rysowane za pomocą nadpisanej metody draw. Pola tej klasy to statystyki jednostki i są dziedziczone przez klasy pochodne. Najważniejsze metody tej klasy to wirtualna metoda attack, której to implementację zawiera każda z klas pochodnych. Klasa zawiera również metody findClosestEnemy oraz isWithinRange, które wykorzystywane są do poprawnego wykrywania przeciwników przez jednostki sojusznicze. Obiekty klas pochodnych różnią się od siebie implementacją metody wirtualnej oraz teksturą i kolorem.
 - meleeAlly – reprezentuje jednostki walczące na bliski dystans. Implementacja metody attack służy do zadania najbliższemu przeciwnikowi w zasięgu ogromnych obrażeń i restart cooldownu jednostki.
 - rangeAlly – reprezentuje jednostki zasięgowe. Metoda attack działa jak w klasie meleeAlly, lecz zamiast zadawania obrażeń tworzy dynamicznie nowy pocisk typu missile o odpowiednich obrażeniach poruszający się w odpowiednim kierunku.
 - supportAlly – jednostki generujące zasoby. Implementacja tej akcji zawarta jest w metodzie attack. Jednostka generuje zasoby w interwałach czasowych, niezależnie od przeciwników i nie zadaje im obrażeń.
 - tankAlly – wytrzymała jednostka pasywna, której implementacja metody attack jest pusta. Jednostka służy tylko do blokowania przeciwników w miejscu przez długi czas.
- allyCard – klasa służąca do reprezentacji kart sojuszniczych jednostek. Posiadają wczytaną teksturę (zdjęcie postawionej jednostki na tle mapy) oraz zmienną string allyType która określa rodzaj jednostki sojuszniczej. Najważniejsze metody tej klasy to getAllyCost, która zwraca koszt roztawienia jednostki z wybranej karty oraz createAlly która tworzy dynamicznie nową jednostkę i rozmieszcza ją na polu slot.
- Enemy – klasa, po której dziedziczą poniższe klasy reprezentujące różne rodzaje przeciwników. Sama klasa enemy nie jest abstrakcyjna, ale jej instancje nie są tworzone. Jej pola to wszystkie statystyki jednostki, takie jak punkty życia czy obrażenia. Metody tej klasy oprócz standardowych „setterów” i „getterów” to metody aktualizujące pozycję obiektu oraz metoda canAttack sprawdzająca czy dana jednostka wroga jest w stanie zaatakować. Klasy pochodne różnią się kolorem rysowanego kształtu, oraz ilością punktów wytrzymałości. Zaimplementowany jest w nich tylko konstruktor ustawiający odpowiednie wartości oraz kolory kształtu:
 - lowTierEnemy – zielona jednostka, najmniejsza wytrzymałość, najwyższa prędkość ruchu
 - midTierEnemy – pomarańczowa jednostka, średnia wytrzymałość i prędkość ruchu
 - highTierEnemy – czerwona jednostka, wysoka wytrzymałość i niska prędkość ruchu

- game – klasa w której metody statyczne obsługują główną pętlę gry i obsługę zdarzeń. Większość metod i funkcjonalności tej klasy opisane jest w sekcji **ogólna struktura programu**.
- Leaderboard – klasa której metody służą do pobierania danych graczy z pliku tekstowego, aktualizowania i sortowania ich oraz wyświetlania. Najbardziej rozbudowana jest właśnie metoda display, w której to zaimplementowane jest tworzenie nowego okna, rysowanie wszystkich danych oraz przycisków do przewijania stron. Metody nextPage oraz previousPage to oddzielne metody inkrementujące lub dekrementujące zmienną currentPage. Określa ona na której stronie aktualnie się znajdujemy.
- Missile – klasa reprezentująca pocisk wystrzeliwany przez jednostki zasięgowe. Jej pola to obrażenia pocisku oraz liczby zmiennoprzecinkowe określające prędkość jednostki i promień jej kształtu. Podobnie jak klasa enemy, missile posiada metodę update aktualizującą pozycję obiektu.
- Player – klasa reprezentująca gracza. Posiada pola takie jak username, totalScore, currentScore czy levelsCompleted. Posiada wiele metod służących do zapisu danych gracza do pliku, czytujących je, sprawdzających istnienie gracza w bazie danych, lub zwracającą wszystkich graczy (dla użytku w klasie Leaderboard).
- popUp – klasa służąca do usprawnienia tworzenia okienek systemowych służących jako alerty z wyświetlanym tekstem, systemową metodą MessageBoxA. Utworzenie obiektu popUp przypisuje odpowiednie zmienne (tekst do przekazania) i wywołuje metodę display, która tworzy okno systemowe z przekazanymi napisami.
- Resources – klasa reprezentująca zasoby dostępne w grze. Dziedziczy po sf::Drawable więc jest możliwe rysowanie całego obiektu, którego reprezentacja graficzna składa się z tekstu oraz obramowania. W klasie nadpisane zostały operatory dodawania oraz odejmowania, w ten sposób wykonuje się odjęcie ilości zasobów przy rozmieszczeniu jednostki w pętli gry.
- Slot – klasa reprezentująca pojedyncze obszar na planszy rozgrywki. Oprócz graficznej postaci posiada ona również jako pole wskaźnik polimorficzny typu ally. W ten sposób postacie sojusznicy są przechowywane w programie. Klasa posiada metody getAlly, setAlly oraz destroyAllyInSlot służące do zarządzania jednostkami. Kolejną ważną metodą jest updateEffectiveRange. Służy do aktualizowania zasięgu postawionej jednostki i jest wywoływana podczas jej postawienia. Zasięg jednostki jest wtedy obliczany w następujący sposób: nowy zasięg dla jednostki w tym polu jest równy: zasięgu jednostki(ilość pól) * szerokość pola. W ten sposób dostosowujemy zasięg do wielkości rysowanych elementów.
- Spawner – klasa służąca tylko i wyłącznie do graficznej reprezentacji miejsc, w których pojawiają się przeciwnicy. Dziedziczy po sf::Drawable i może być rysowana, parametry kształtu ustawiane są w konstruktorze.
- textField – klasa, której obiekty to pola tekstowe służące do wprowadzania napisu, a potem do operowania na nich. Zaimplementowanie tej klasy jest wymagane, ponieważ SFML sam w sobie nie dostarcza tego typu obiektów. Wykorzystywana w programie jako pole do wprowadzania nazwy gracza. Najważniejsza metoda handleInput służy do ustawiania tekstu gdy aktywny jest Event przekazany jako parametr(jeśli Event odpowiada wprowadzaniu tekstu przez klawiaturę).

- TextureManager – nie-autorska klasa która zarządza teksturami. Dzięki statycznym metodom możliwe jest uprzednie wczytanie tekstur do programu, a następnie dostęp do nich przez strukturę mapy. Link do githuba autora tej klasy został umieszczony w **literaturze**. W ramach tego projektu, klasa została jedynie zrefaktoryzowana z plików .h oraz .cpp na moduł.
- Wave – klasa, której statyczne metody służą do generowania fal przeciwników w zależności od aktualnie rozgrywanego poziomu. Klasa została dokładniej opisana w sekcji **ogólna struktura programu**.

4.4. Ogólna struktura programu

Wraz ze startem programu w funkcji main wywoływana jest metoda statyczna gameMenu klasy game. Tworzone w niej jest okno menu programu, tworzone są odpowiednie przyciski i pola tekstowe, a następnie inicjalizowana jest pętla programu, która rysuje obiekty i obsługuje zdarzenia, takie jak kliknięcia. Dodatkowo wczytywane są tekstury przycisków wykorzystywane w oknie Leaderboard. Obiekt klasy game nigdy nie jest tworzony, składa się ona tylko ze zmiennych i metod statycznych. Kliknięcie każdego z przycisków w menu uruchamia inną metodę statyczną klasy game (rysującą odpowiednie okno), z wyjątkiem przycisku Leaderboard, który to tworzy nowy obiekt klasy Leaderboard, a następnie uruchamia metodę display tej klasy, rysującą okno i wyświetlającą dane. W menu wykorzystywane są również metody loginPlayer oraz registerPlayer. Po kliknięciu w odpowiednie przyciski są one wywoływane. Działanie pierwszej metody ogranicza się do stworzenia nowego obiektu typu player, wywołanie statycznej metody klasy player sprawdzającej poprawność wpisanej nazwy gracza, a następnie zwrócenie obiektu. Jeśli nie wszystkie warunki poprawności są spełnione zwrócona nazwa gracza jest pusta lub równa napisowi „null” zwracanemu gdy użytkownika nie znaleziono w bazie danych, wtedy logowanie nie następuje (klauzula if w obsłudze zdarzeń – kliknięć przycisku myszy). Aktualnie zalogowany gracz jest przechowywany w zmiennej statycznej currentPlayer klasy game. Metoda register działa podobnie. Stworzony obiekt zawierający wprowadzoną nazwę gracza jest przekazywany jako argument do metody klasy player. Tam następuje sprawdzenie, czy użytkownik o danej nazwie już istnieje w bazie danych. Jeśli nie, rejestracja przebiega pomyślnie. Oprócz tego w tej metodzie są dwie metody saveSettings oraz loadSettings, które zapisują i wczytują ustawienia z okienka Options do pliku tekstowego w reprezentacji liczbowej.

Po rozpoczęciu rozgrywki przyciskiem Adventure bądź Survival uruchamia się najważniejsza metoda gameLoop. W trybie Adventure dzieje się to pośrednio, gdyż najpierw wywoływana jest metoda chooseLevel. W tym okienku to który przycisk zostanie kliknięty wpływa na wartość zmiennej currentLevel. Określa ona następnie ilość wierszy w planszy gry, ilość wygenerowanych kart jednostek sojusznich, oraz wpływa na to czy punkty rankingowe mogą być zliczane (mogą tylko w trybie Survival, gdzie currentLevel = 7). W trybie Survival metoda gameLoop wywoływana jest bezpośrednio ze stałą wartością currentLevel. Wraz ze startem metody gameLoop dzieje się wiele rzeczy. Tworzone są wszystkie obiekty graficzne. Wczytywane są również tekstury kart sojusznich. Następuje wyzerowanie wszystkich obiektów zliczających czas potrzebnych do obliczania prawidłowych interwałów czasowych (w generowaniu przeciwników oraz zasobów).

Tworzone są zmienne potrzebne między innymi do ustalenia początkowej ilości zasobów, czy reprezentacji aktualnie klikniętej karty.

Następnie generowane jest okno rozgrywki i rozpoczyna się główna pętla gry.

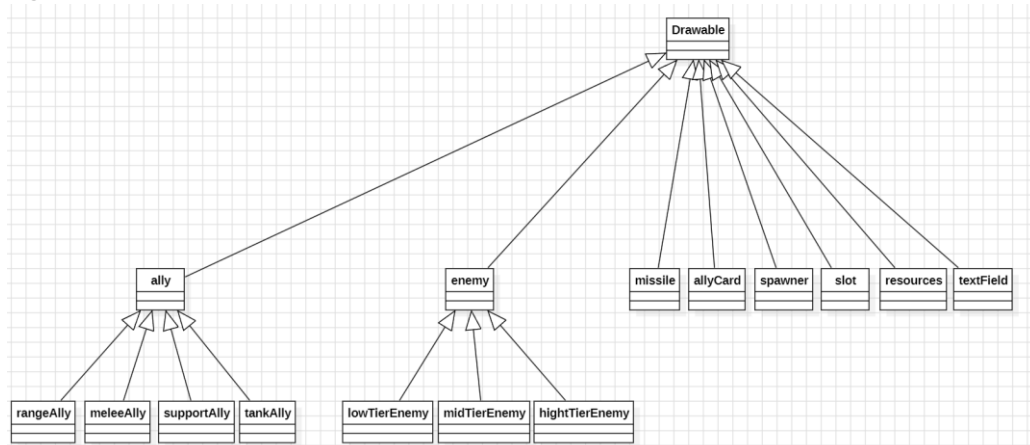
To w niej zaimplementowana jest obsługa kliknięć myszy potrzebna do rozstawiania jednostek.

W tej pętli znajduje się również implementacja wszystkich zdarzeń, takich jak rysowanie jednostek, wykonywanie ich akcji w odstępach czasowych, oraz co najważniejsze – wszystkie rodzaje kolizji między obiektami. Znajduje się tu również warunek końca gry (dla trybu Adventure) – jeśli dana fala przeciwników została pokonana pamięć zaalokowana dynamicznie jest zwalniana, a następnie ilość zaliczonych przez gracza poziomów zostaje zwiększona o 1.

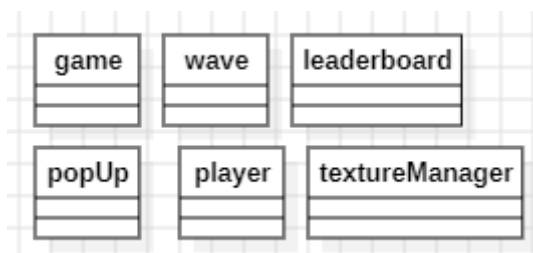
Zarówno tryb Adventure jak i Survival uruchamiają tę samą pętlę gry. Różnią się one jednak wywoływaniem w tej metodzie statycznych metod klasy wave. Odpowiednia fala jest wywoływana zależnie od wyboru case w instrukcji switch. Instrukcja ta wywołuje się sama w pętli gry, a wybrany przypadek zależy od wartości zmiennej currentLevel. Klasa wave składa się ze zmiennych i metod statycznych. Zmienne są w większości typu bool i określają czy dana „fala” przeciwników została pokonana. Pozostałe zmienne to obiekty mierzące czas. Same metody odpowiadają kolejnym poziomom (np. createWave1 służy do obsługi fal w pierwszym poziomie... i tak dalej). Metody składają się ze zbioru instrukcji warunkowych, gdzie mierzony jest czas. Jeśli określony czas upłynął, a odpowiednie zmienne są ustawione na true, przeciwnicy są tworzeni i zwracani do głównego wektora przeciwników, który przekazany został do metody jako argument przez referencję. Jeśli fala została zakończona, zmienna levelCompleted zostaje ustawiona na true. Dostęp do tej zmiennej statycznej ma klasa game, a warunek końca w pętli gry zależy od tej zmiennej.

W trybie Survival fale nie są tak specyficznie zaprojektowane. Zamiast tego przeciwnicy generują się nieskończenie długo, aż nie nastąpi porażka gracza. W metodzie createSurvivalWave po zmierzeniu czasu sprawdzany jest warunek czy odstęp czasu pomiędzy pojawieniem się przeciwników jest większy od minimum. Jeśli tak, ta wartość dla następnej iteracji jest zmniejszana o 5%. Następnie instrukcje warunkowe sprawdzają ilu przeciwników zostało dotychczas utworzonych, aby następny utworzony byładanego rodzaju (jak opisano w **Specyfikacji zewnętrznej**).

4.5. Diagram klas



Zrzut ekranu 15: Diagram klas z dziedziczeniem



Zrzut ekranu 16: Diagram klas – pozostałe klasy

4.6. Aktywności – opis słowny

Po uruchomieniu programu użytkownik znajduje się w menu. Z jego poziomu może uruchomić jedno z okien poprzez kliknięcie odpowiednich przycisków:

- Options
- Help
- Leaderboard

Wyście z tych okien skutkuje powrotem do menu. W menu użytkownik może się również zarejestrować lub zalogować korzystając z przycisków kolejno Register oraz Login. Po zalogowaniu oraz skonfigurowaniu ustawień użytkownik może rozpocząć rozgrywkę klikając przycisk Survival lub Adventure. W trakcie rozgrywki użytkownik może rozmieszczać sojusznice jednostki na planszy lewym przyciskiem myszy. Rozgrzywka trwa tak długo, aż nie nastąpi pokonanie wszystkich przeciwników lub porażka. Wtedy tworzone jest odpowiednie okno ilustrujące wygraną/porażkę. Po jego zamknięciu użytkownik przechodzi z powrotem do menu.

4.7. Szczegółowy opis zagadnień obiektowych

- W projekcie dziedziczenie oraz polimorfizm zaimplementowane są w kilku miejscach. Wszystkie klasy dziedziczące po `sf::Drawable` są możliwe do rysowania poprzez nadpisanie abstrakcyjnej metody `draw`. Klasy te są widoczne na **diagramie klas**. Klasa `ally` jest abstrakcyjna. Dziedziczące po niej klasy pochodne nadpisują wirtualną metodę `attack`, w celu wykonywania różnych akcji w zależności od jednostki. Polimorfizm jest tutaj potrzebny w celu wykorzystania polimorficznego wskaźnika typu `ally`, między innymi w definicji klasy `slot`. Wskaźnik na obiekt klasy pochodnej po `ally`, jest bowiem jednym z pól klasy `slot`, co jest przykładem kompozycji. Obiekty klasy `enemy` nie są tworzone, mimo że klasa ta nie jest abstrakcyjna, ponieważ nie musi być. Na chwilę obecną różne klasy pochodne po `enemy` nie różnią się od siebie funkcjonalnością, a jedynie wartościami pól. Nie ma więc potrzeby tworzenia i nadpisywania metod wirtualnych. Wszystkie te zależności i nazwy poszczególnych klas widoczne są na **diagramie klas** oraz w **opisie klas**.
- W projekcie zostały użyte również techniki obiektowe – zagadnienia tematyczne wykorzystywane na laboratorium z najnowszych standardów języka C++. Postawiono na użycie czterech technik:
 - Użycie modułów zamiast `#include`. Pozwala to na ulepszenie bezpieczeństwa i niezawodności oraz przyspieszenie czasu kompilacji kodu. Zdecydowana większość plików w projekcie to pliki modułowe.
 - Biblioteka `<regex>` - wyrażenia regularne użyte zostały w projekcie w jednym miejscu: podczas sprawdzania poprawności nazwy użytkownika przy rejestracji
 - Biblioteka `<filesystem>` - w projekcie wykorzystywane są zmienne typu **`path`**, które dostarcza ta biblioteka. Służą przechowywaniu ścieżki do pliku. Jest to wygodna alternatywa dla umieszczania ścieżki w zmiennej `string`.
 - Biblioteka `<ranges>` - zdecydowanie najbardziej funkcjonalna biblioteka w tym projekcie. Jest wykorzystywana między innymi w trakcie porównywania nazw użytkownika (`ranges::equal`) – porównywanie zakresów. Drugim kluczowym miejscem jest metoda `findClosestEnemy` klasy `ally`. Użyta została funkcja `ranges::sort` w celu posortowania wektora przeciwników, a także adaptery zakresowe `views::filter` wraz z symbolem potoku `|` w celu przefiltrowania wektora odpowiednimi warunkami.

5. Testowanie

Projekt był testowany podczas dodania każdej nowej funkcjonalności. Nie były to testy jednostkowe czy Test Driven Development. Gra była rozgrywana przez autora wiele razy co skutecznie pozwoliło na wykrycie dużej ilości błędów (rzecz jasna nie jest to w stanie zagwarantować całkowitej niezawodności programu, ale pozwala na wykrycie większości oczywistych problemów).

W trakcie procesu testowania programu potkano wiele błędów. Poniżej wymieniono kilka z nich oraz sposób poradzenia sobie z ich rozwiązaniem:

- Błąd z niepoprawnym zasięgiem sojusznicznych jednostek – zauważono, że niezależnie od ustawionej wartości zasięgu jednostki (float określający ilość pól, np. $1.5 = 1.5$ pola zasięgu), zasięg jej ataku pozostaje stały (1 pole). Na rozwiązanie natrafiono, gdy w ramach testu ustawiono ten zasięg na wyższą wartość (2.5). Okazało się, że wtedy faktyczny zasięg jednostki wynosił około 2.0. Po krótkim namyśle zdano sobie sprawę, że getter zwracający zasięg jednostki jest typu int, podczas gdy sama zmienna jest typu float. Prowadziło to do nieumyślnego zaokrąglania wartości do liczby całkowitej. Problem naprawiono, zmieniając typ gettera na float.
- Błąd związany z nieporuszaniem się przeciwnika po zniszczeniu sojusznicznej jednostki. Przeciwnicy stają w miejscu gdy zaczynają atakować sojusznika, a ich prędkość zostaje przywrócona gdy tę jednostkę zniszczą. Podczas gdy wielu przeciwników atakowało tę samą jednostkę sojuszniczą czasem zdarzał się błąd, przez który jeden z przeciwników poprawnie przywracał swoją prędkość, a drugi - nie wiedzieć czemu – stał dalej w miejscu. Z problemem poradzono sobie poprzez dodanie jednej nowej pętli iterującej przez pola slot, a następnie przez przeciwników. Jeśli pole jest wolne (nie ma na nim sojusznicznej jednostki, `slot.isOccupied() = false`), sprawdzana jest kolizja pomiędzy tym polem a każdym przeciwnikiem. Następnie prędkość ruchu każdego przeciwnika na niezajętym polu zostaje przywracana do wartości domyślnej dla tej jednostki.
- Błąd związany z rozmieszczaniem jednostek. Czasami, gdy rozstawiano jednostkę jednego typu, jej koszt nie wynosił tyle ile powinien, przez co ilość zasobów mogła wynosić mniej niż 0. Okazało się, że przez niedopatrzenie autora zmienna statyczna określająca koszt jednostki była inicjowana w abstrakcyjnej klasie bazowej, a następnie jej wartość nadpisywano w klasach pochodnych. Był to oczywisty błąd, gdyż zmienna statyczna mogła być współdzielona przez klasy pochodne, ale jej wartość była również współdzielona przez te klasy. Prowadziło to często do dziwnego zachowania i nieoczywistych błędów. Błąd naprawiono usuwając tę zmienną z procesu dziedziczenia jak i z klasy bazowej i inicjalizując ją od zera w każdej klasie pochodnej. Dzięki temu każda z klas posiadała jej własną wartość, niezależną od innych.

Literatura

- <https://www.sfml-dev.org/learn.php>
- <https://www.youtube.com/watch?v=JlAd3X3PX6o&list=PLk6mhiZKpyW4KRTZc8sc0aYOLFmTSLA7r>
- <https://github.com/netpoetica/sfml-texture-manager>
- Wykłady z przedmiotu Programowanie Komputerów