Dokumentacja projektu

# **Modyfikacja Galagi**

**Autor:** Daniel Cogiel

Politechnika Śląska
15 czerwca 2022

# Spis treści

# Analiza tematu

## 1.1 Ogólne założenia

Stworzony projekt jest modyfikacją gry Galaga (`https://pl.wikipedia.org/wiki/Galaga`). Statek kosmiczny może poruszać się po całym ekranie i strzelać do spadających z góry obiektów (przykładowo asteroid). Za zniszczenie obiektu gracz otrzymuje odpowiednią dla danego typu obiektu liczbę punktów. Gdy gracz zderza się z obiektem, traci daną ilość życia. Celem gry jest zdobycie jak największej ilości punktów, zanim życie gracza spadnie do zera. W grze przewidziane jest 5 typów różnych przeciwników i jeden bonus (*astronauta*), każdy z własną teksturą i charakterystycznymi dla siebie parametrami i sposobem poruszania się.

## 1.2 Szczegółowy opis użytych bibliotek

Wymienione wcześniej rozwiązania omawiane na zajęciach laboratoryjnych zostały wykorzystane w projekcie w następujący sposób:

- Biblioteka **REGEX** - używana jest do sprawdzania, czy nazwa podawana przez gracza jest poprawna.
- Biblioteka **Filesystem** - w obiektach **std::filesystem::path** przechowywane są ścieżki do tekstur, plików audio i czcionek, które następnie używane są podczas wczytywania tych zasobów.
- Biblioteka **Ranges** - używana jest do szybkiego sortowania danych, niektóre metody używają także pętli **for_each** biblioteki

- Prosta współbieżność (głównie **std::thread** i **join()**) - stosowana jest podczas wczytywania danych w konstruktorze klasy **Gra**

Do stworzenia warstwy graficznej projektu została użyta biblioteka graficzna **SFML**. Biblioteka ta umożliwia sprawne i proste renderowanie obiektów na ekranie i dostarcza różne użyteczne funkcjonalności, takie jak przykładowo **sf::Clock** pozwalający na pomiar czasu w czasie działania programu.

## 1.3 Podział na klasy

Zgodnie z paradygmatem programowania obiektowego różne funkcjonalności programu zamknięto w odpowiednich klasach. Główną klasą reprezentującą grę, przechowującą wszystkie jej zasoby i implementującą funkcjonalności jest klasa **Gra**.

# Specyfikacja zewnętrzna

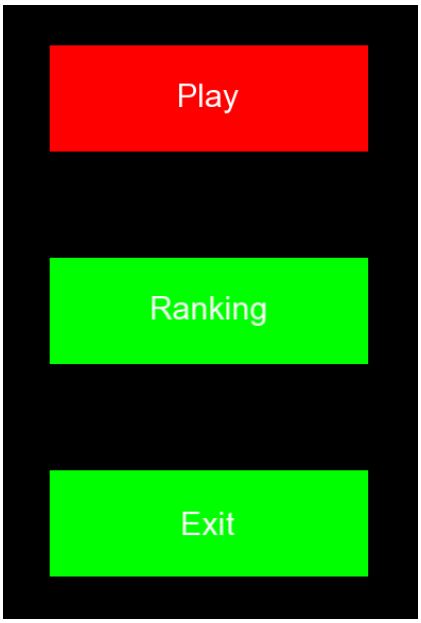
## 2.1 Interfejs i schemat działania

Program startuje poprzez uruchomienie pliku wykonywalnego **.exe** bez podawania parametrów. Po uruchomieniu użytkownik jest pytany o podanie swojej nazwy, której poprawność jest weryfikowana przez **REGEX** (2.1). Następnie po wczytaniu zasobów wyświetla się okno, a w nim menu główne gry (2.2). Menu to ma 3 opcje: *Play* uruchamiającą grę, *Ranking* wyświetlającą ranking (2.3) oraz *Exit* kończącą działanie programu. Po menu można poruszać się za pomocą strzałek bądź klawiszy W i S. Po wybraniu opcji *Play* wyświetlana jest krótka instrukcja dla gracza wraz ze schematem sterowania (2.4). Po naciśnięciu dowolnego klawisza rozpoczyna się właściwa gra (2.5). Po zakończeniu gry (dzieje się to, gdy życie gracza spada poniżej zera) wyświetlany jest ekran *Game Over*. Po wciśnięciu dowolnego klawisza podczas wyświetlania tego ekranu następuje powrót do menu głównego. Podczas gry gracz może zrobić sobie przerwę, wciskając w dowolnym momencie klawisz *Escape*; po wciśnięciu tego klawisza wyświetla się menu pauzy z 3 opcjami: *Continue* wznawiającą grę, *Main Menu* powracającą do menu głównego oraz *Exit* kończącą grę (2.6).

## 2.2 Zrzuty ekranu

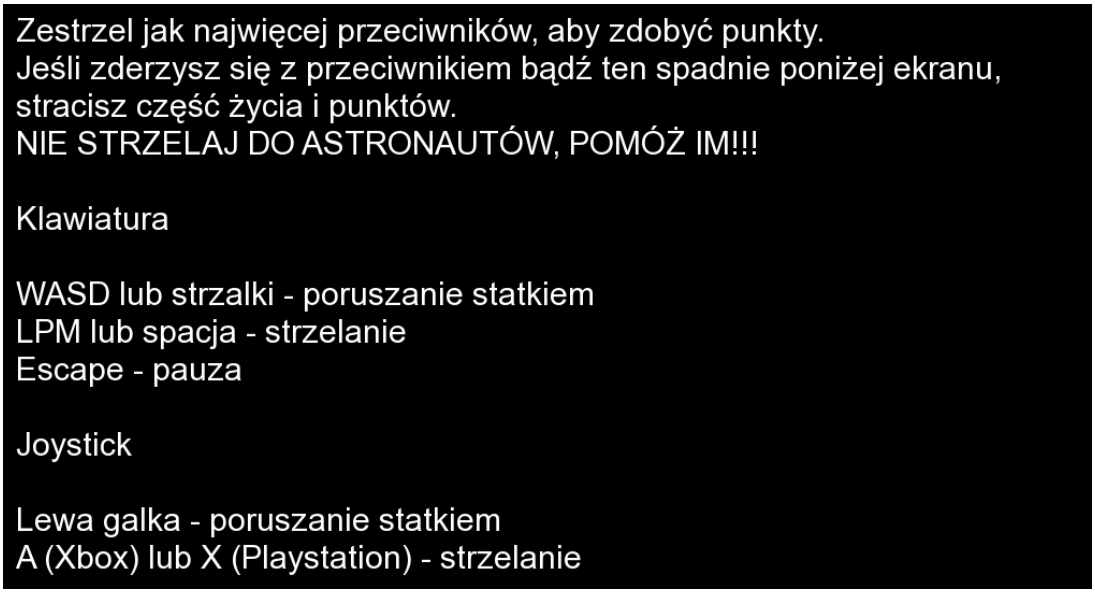
```
Nazwa musi zaczynac sie od litery i moze zawierac do 20 znakow (liczby, podkreslenia i myslniki)
Wprowadz nazwe gracza:
```

Grafika 2.1: Zapytanie o nazwę użytkownika

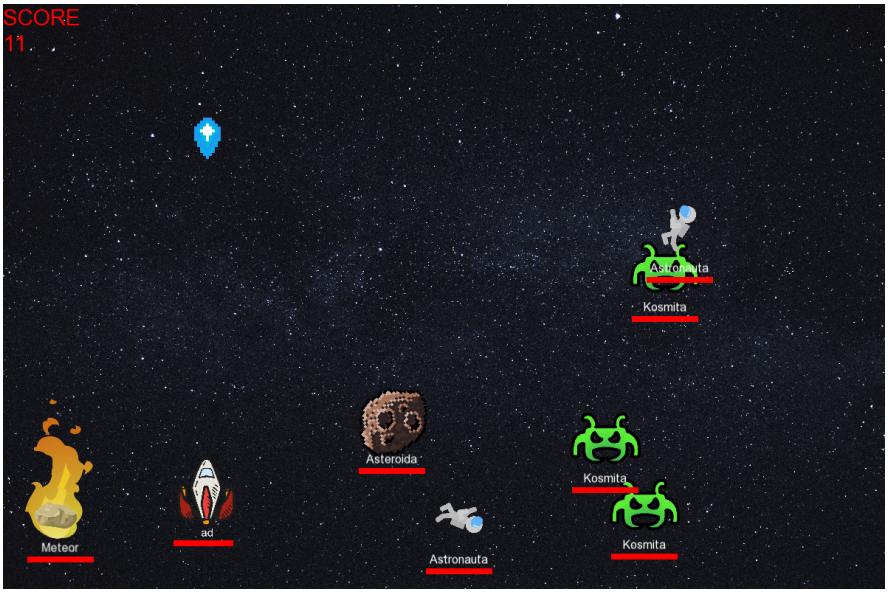



Grafika 2.2: Menu główne

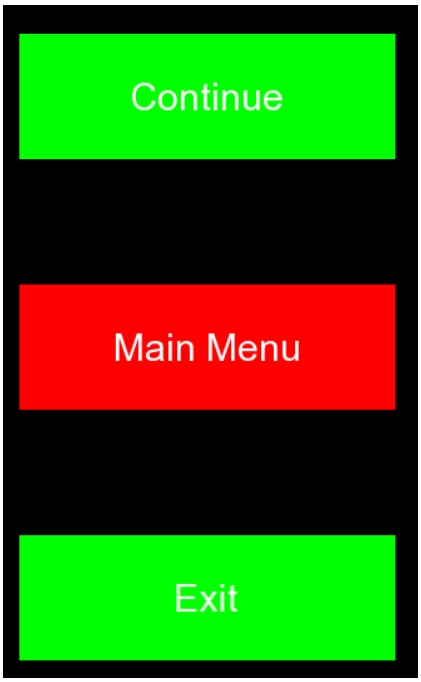

Grafika 2.3: Ranking




Grafika 2.4: Instrukcja dla gracza




Grafika 2.5: Zrzut ekranu gry




Grafika 2.6: Menu pauzy

# Specyfikacja wewnętrzna

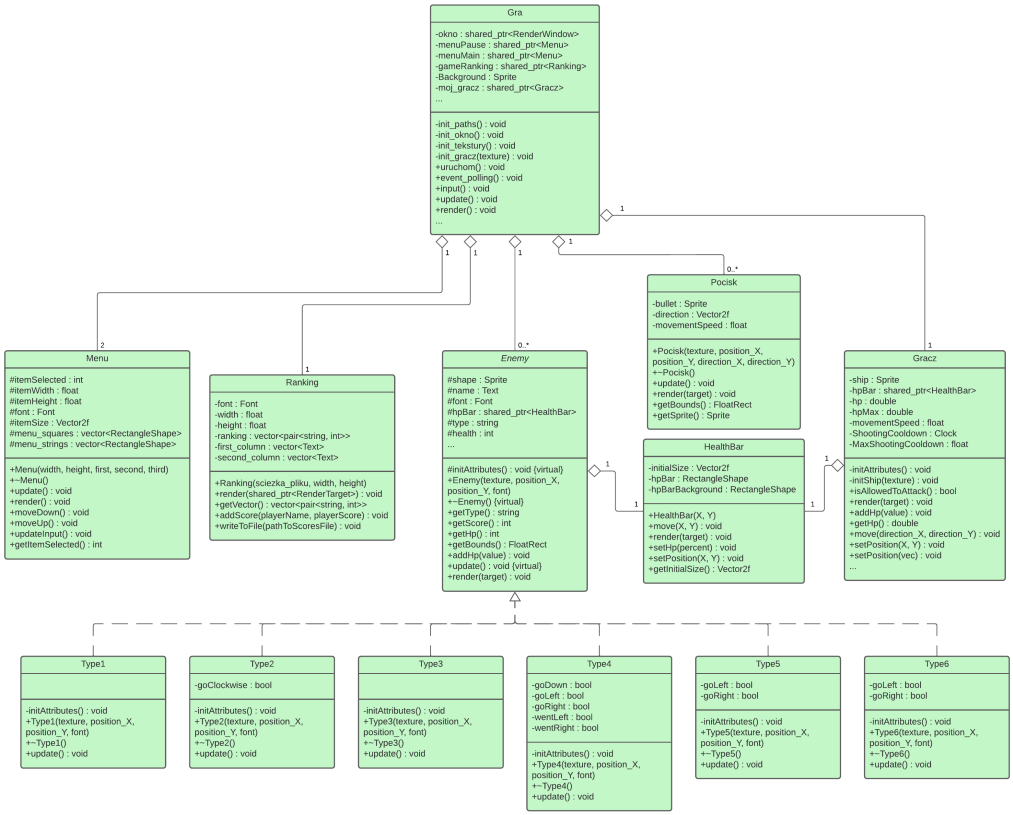
## 3.1 Istotne metody zaimplementowanych klas

- **Gra::uruchom()** - zawiera główną petlę gry, w której aktualizowane i renderowane są obiekty
- **Gra::update()** - aktualizuje obiekty i wyłapuje zdarzenia w oknie
- **Gra::render()** - wyświetla obiekty na ekranie
- **Gra::Gra()** - inicjalizuje obiekt gry i wczytuje wszystkie zasoby
- ***Enemy::update()*** - metoda czysto wirtualna, klasy pochodne implementują w niej różne sposoby poruszania się przeciwników na ekranie

## 3.2 Szczegółowy opis klas i struktur danych

Szczegółowa dokumentacja zaimplementowanych klas i struktur danych znajduje się w załączniku "Opis_klas_i_struktur_danych.pdf".

## 3.3 Diagram hierarchii klas




Grafika 3.1: Diagram hierarchii klas programu

## 3.4 Ogólny schemat działania

Na początku działania programu tworzony jest obiekt klasy **Gra** i jest wywoływana jego metoda **uruchom()**. Metoda ta funkcjonuje przez cały czas działania programu, do momentu kiedy zostanie zamknięte okno gry. Metoda **uruchom()** korzysta z dwóch innych metod klasy **Gra**: **update()** i **render()** o funkcjonalnościach opisanych w punkcie 3.1. W czasie działania pętli aktualizowane są obiekty na ekranie, wychwytywane są zdarzenia (np. zamknięcie okna bądź wciśnięcie Escape), aktualizowany jest interfejs. Punkty zdobyte przez gracza są wczytywane do pliku *Scores.txt*; dzieje się to, gdy gracz zamknie okno, wróci do menu głównego podczas gry lub zakończy grę w naturalny sposób (poprzez spadnięcie poziomu życia poniżej zera). Zachowanie programu jest uzależnione od stanu, w którym znajduje się gra. Stany reprezentowane są poprzez zmienne **bool**. Stanów jest w sumie 6: *statePlay*, *statePaused*, *stateMainMenu*, *stateRanking*, *stateInstruction* oraz *stateGameOver*. W zależności od danego stanu, w którym znajduje się gra, wciśnięte klawisze przez gracza powodują dany efekt, a na ekranie wyświetlają się odpowiednie obiekty. Warto również wspomnieć o metodzie czysto wirtualnej **update()** klasy ***Enemy***. Klasy pochodne **Type1**, **(...)**, **Type6** implementują tę metodę, definiując w niej swój sposób poruszania po ekranie.

# Testowanie i uruchamianie

## 4.1 Przypadki ekstremalne

W grze zostały przetestowane ekstremalne przypadki takie jak:

- **Brak pliku z teksturą** - na ekranie wyświetlany jest odpowiedni błąd, następnie gra uruchamia się w sposób naturalny, lecz miejsce, w którym miała znaleźć się tekstura podczas gry, jest puste
- **Brak pliku dźwiękowego** - na ekranie wyświetlany jest odpowiedni błąd, następnie gra uruchamia się w sposób naturalny, lecz bez ścieżki dźwiękowej
- **Brak pliku z czcionką** - na ekranie wyświetlany jest odpowiedni błąd, następnie gra uruchamia się w sposób naturalny, lecz w miejscach napisów pojawia się pojedyncza kropka (".")
- **Brak pliku *Scores.txt*** - gra uruchamia się i działa w naturalny sposób, a gdy nadchodzi moment, w którym należałoby zapisać wynik gracza, tworzony jest nowy plik *Scores.txt*
- **Pusty plik *Scores.txt*** - gra uruchamia się i działa w normalny sposób, pusty plik *Scores.txt* nie powoduje żadnych błędów podczas wyświetlania rankingu

## 4.2 Wycieki pamięci

Ze względu na specyfikę biblioteki SFML standardowe narzędzia sprawdzające obecność wycieków pamięci w programie takie jak **_CrtDumpMemoryLeaks()** nie dają poprawnego wyniku. Zasadne jest jednak założenie, że program nie powoduje wycieków pamięci, gdyż stosowane są w nim dobre praktyki programowania - używane są inteligentne wskaźniki **shared_ptr()**, które dealokują pamięć samodzielnie.

## 4.3 Wnioski

Gra działa poprawnie i płynnie nawet w przetestowanych ekstremalnych przypadkach. Podczas projektowania kładzony był nacisk na przejrzystość i zrozumiałość zasad i mechanik gry dla użytkownika. Udało się to osiągnąć poprzez częstą interakcję z użytkownikiem i wyświetlanie instrukcji na ekranie. Biblioteka **SFML** w znaczący sposób ułatwiła projektowanie programu, ze względu na jej łatwe do zrozumienia funkcje oraz przydatne mechanizmy takie jak **sf::Clock**, łatwe wczytywanie tekstur z pliku i przypisywanie ich do obiektów etc.