# **Języki Programowania Obiektowego**

Dokumentacja Projektowa

**Gra komputerowa typu “Shooter 2D”**

**Spis treści**

[**I. Wstęp 3**](#_heading=h.vcq2vr58ev22)

[**III. Funkcjonalność 6**](#_heading=h.q8evvafxpzq1)

[**IV. Projekt techniczny 8**](#_heading=h.8sdx5ed27v1p)

[**V. Opis realizacji 10**](#_heading=h.rrdzwmub2ey1)

[**VI. Testy 10**](#_heading=h.37k594j1kzs9)

[**VII. Podręcznik użytkownika 11**](#_heading=h.vl2sk8u7mjmt)

[**Bibliografia 14**](#_heading=h.dea0g6bcszld)

# **Wstęp**

Dokument dotyczy opracowania gry komputerowej 2D. Celem projektu ma być zapewnienie użytkownikowi rozrywki w postaci rozgrywki komputerowej. Jako dodatkowe cele wyznaczono naukę i rozwój osobisty w dziedzinie programowania.

Jednym z takich celów jest raycasting 2D. Poznanie mechanik działania tej idei oraz wykonywanie podstawowych operacji na jej bazie.

# 

1. **Plan projektu**
   1. **Idea działania gry**

Pierwszym krokiem rozpoczynającym projekt będzie działanie gry. Ma ona polegać na poruszaniu się graczem na mapie, na której będą znajdować się przeciwnicy różnego typu. Zadaniem gracza będzie dotrzeć do danego punktu, aby ukończyć grę. Przy poruszaniu się po mapie gracz musi unikać zasięgu wzroku przeciwników, gdy zostanie wykryty gra kończy się porażką. Użytkownik będzie miał możliwość eliminacji wrogów.

* 1. **Stworzenie katalogów oraz konfiguracja bibliotek**

Projekt zostanie podzielony w następujący sposób:

-katalog build/, w tym katalogu będą przechowywane wszystkie pliki utworzone w procesie budowania projektu przez narzędzie CMake

-katalog examples/, będzie zawierał jeden plik źródłowy, który posłuży jako main

-katalog include/, będzie zawierał kolejny katalog Classes/, w którym umieszczone zostaną wszystkie pliki nagłówkowe poszczególnych klas

-katalog src/, będzie zawierał wszystkie pliki źródłowe poszczególnych klas

Po stworzeniu powyższej struktury do każdego katalogu zostanie dodany plik CMakeLists.txt, ponadto w katalogu głównym również zostanie dodany taki plik (będzie służył jako główny CMakeLists.txt). Następnie wszystkie pliki związane z CMake’iem zostaną skonfigurowane. Dołączona zostanie biblioteka SFML oraz GoogleTest.

* 1. **Wstępne rozplanowanie podstawowych klas**

Następnie powstanie wstępny plan podziału funkcji do różnych klas. Dla zrozumienia działania raycastingu 2D powstanie prosta struktura projektowa. Będzie zawierać dwie klasy Game oraz Ray. W kolejnym kroku obserwowane będzie zachowanie obiektów klasy Ray w różnych sytuacjach.

* 1. **Mapa, przeciwnicy**

Kolejnym krokiem będzie zaplanowanie struktury mapy i przeciwników. Wstępny plan ich zachowania.

* 1. **Implementacja klas**

Następnie zostanie zaimplementowane działanie poszczególnych klas. Oraz utworzenie klas potrzebnych do działania np.przeciwników. Jedną z takich klas będzie klasa Path.

* 1. **Dodatkowe klasy**

Dodanie dodatkowych klas tj. Bullet, Star, FinishStar.

* 1. **Logika rozpoczęcia i zakończenia gry**

Implementacja logiki do wyświetlania ekranu startowego, końcowego i wyświetlanie odpowiednich komunikatów. Dodana zostanie również możliwość resetu gry w przypadku sukcesu bądź porażki.

# **Funkcjonalność**

* 1. **Game**

Podstawowa klasa realizująca podstawowe funkcje gry. Ma pełnić funkcję silnika gry. Zawiera wskaźniki na obiekty mapy, gracza, przeciwników, pocisków, gwiazdek. Posiada również obiekty okna, wydarzeń, myszki, tekstów, czcionek z biblioteki SFML. Inne zmienne konieczne do logiki gry zostały przedstawione w rozdziale Projekt techniczny. Najważniejsze funkcje to update() i render(), które są odpowiedzialne odpowiednio za uaktualnianie obiektów innych oraz ich wyświetlanie(w tych funkcjach wołane są korespondujące funkcje update i render dla poszczególnych obiektów). Pozostałe funkcje realizują logikę gry (start/zakończenie/reset) oraz wystrzeliwanie pocisków, sprawdzanie kolizji pocisków z mapą/przeciwnikami, zbieranie gwiazdek czy usuwanie obiektów.

* 1. **Player**

Klasa Player ma odzwierciedlać gracza. Wizualizacja została zaimplementowana jako kształt koła. Posiada m.in. celownik, który jest pojedynczym ray’em skierowanym w kierunku myszki, ray’e skierowane w każdym kierunku od gracza(360 stopni) - używane do sprawdzanie kolizji z mapą. Zaimplementowano możliwość poruszania się za pomocą klawiszy WSAD na klawiaturze.

* 1. **Enemy**

Klasa bazowa dla klas Guard, Counterterrorist, Specialforce. Jest realizacją przeciwnika. Posiada funkcje pure virtual dla funkcji inicjalizujących zmienne tej klasy, klasy dziedziczące posiadają własną implementację tych funkcji. Posiada obiekt klasy Path, który definiuje ścieżkę jaką ma się poruszać przeciwnik. Sprawdza czy gracz został wykryty przez danego przeciwnika.

* 1. **Guard, Counterterrorist, Specialforce**

Klasy pochodne z klasy Enemy. Dziedziczą wszystkie funkcje klasy Enemy oraz każda ma inną implementację funkcji inicjalizujących zmienne, ponieważ każda z tych klas posiada inne parametry tj. ścieżka, ilość zdrowia, prędkość poruszania się, długość ray’ów.

* 1. **Map**

Klasa realizująca mapę. Posiada sf::VertexArray, czyli tablicę punktów między którymi mają zostać wyświetlone linie pełniące funkcję ścian.

* 1. **Path**

Klasa realizująca funkcję ścieżki dla przeciwników. W zależności od podanego parametru do konstruktora przyjmuje inne punkty reprezentujące ścieżkę (inna dla GUARD\_1, inna dla SPECIALFORCE). Oprócz punktów, między którymi przeciwnicy mają się poruszać, posiada również w jakiej kolejności mają się przemieszczać, początkowy kierunek, początkową pozycję.

* 1. **Ray**

Klasa reprezentująca promień (ray). Jej działanie jest proste, rysuje linie między dwoma punktami w zadanym kierunku o danej długości. Posiada funkcje do sprawdzania kolizji z mapą oraz rotacji.

* 1. **Bullet**

Klasa reprezentuje pocisk. W trakcie wystrzelenia pocisku rotuje prostokąt w zadanym kierunku (przekazany kierunek celownika). Sprawdzanie kolizji z przeciwnikami oraz z mapą są sprawdzane w klasie Game.

* 1. **Star**

Klasa reprezentująca gwiazdkę. Gwiazdki są “znajdźkami”, które gracz może zbierać (ale nie musi), jest to dodatek do gry.

* 1. **FinishStar**

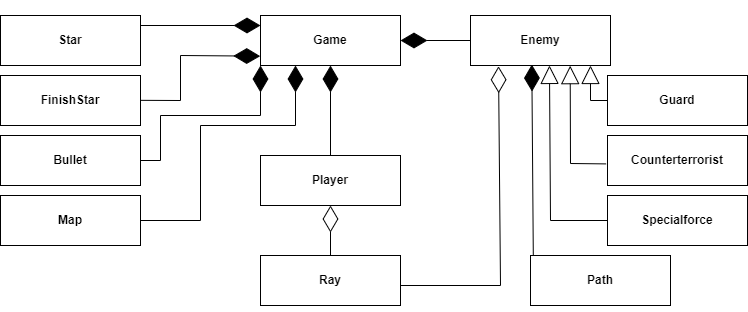
Klasa reprezentująca końcową gwiazdkę, którą gracz musi zebrać aby ukończyć grę z sukcesem.

# **Projekt techniczny**

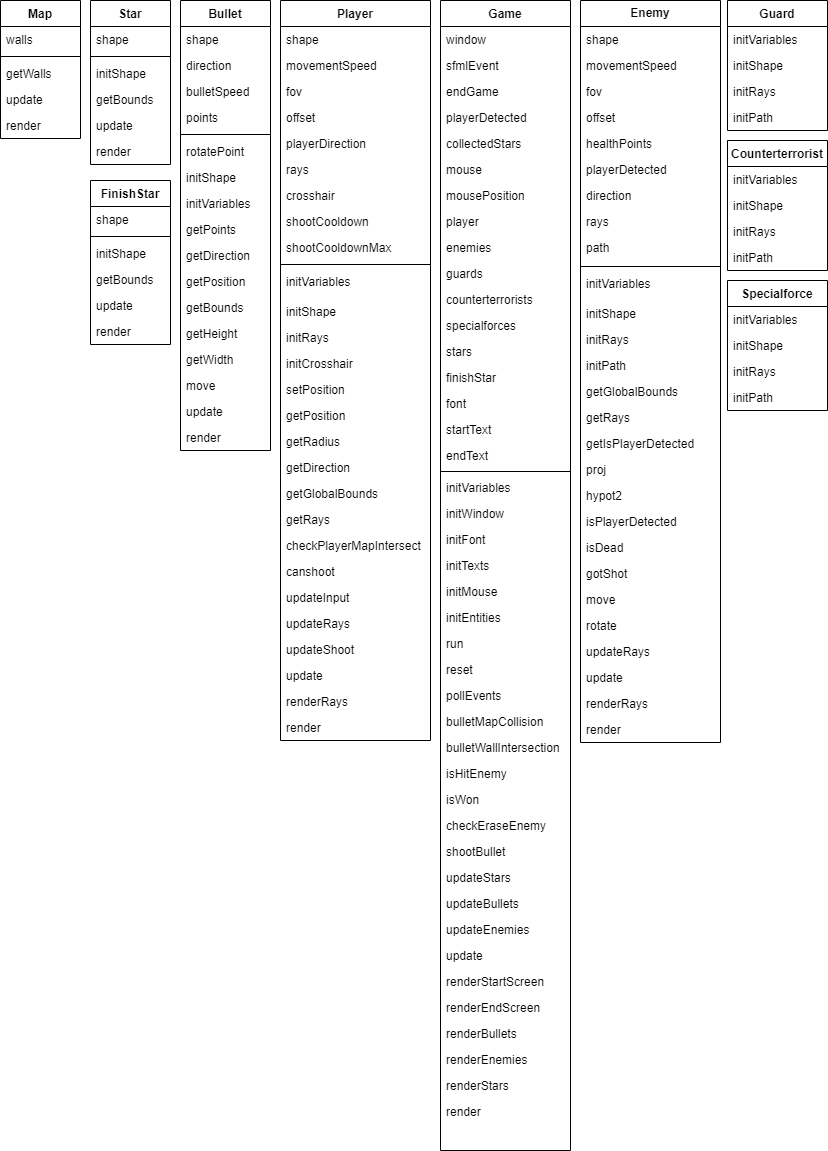
Ze względu na sporą ilość klas umieszczenie wszystkich klas wraz ze zmiennymi i funkcjami w jednym diagramie jest trudnym zadaniem. Z tego powodu podzielono diagram w następujący sposób:

-diagram przedstawiający strukturę projektową - bez nazw zmiennych i funkcji

-diagram przedstawiający każdą klasę osobno z nazwami zmiennych i funkcji



*Rys.1 Diagram przedstawiający zależności pomiędzy klasami*



*Rys.2 Diagram przedstawiający każdą klasę osobno z wymienionymi zmiennymi i funkcjami*

# **Opis realizacji**

Do realizacji projektu użyto:

-IDE: Visual Studio Code

-system operacyjny: Windows 11 Pro x64

-kompilator: mingw w64

-zewnętrzne biblioteki: SFML 2.6.1

-inne narzędzia: CMake, git, debugger VS Code

-github

# **Testy**

Wszystkie bugi w kodzie były wykrywane i sprawdzane za pomocą debuggera wbudowanego w używane IDE, w tym przypadku Visual Studio Code. Najpoważniejszymi bugami były:

-niepoprawne obracanie ray’ów w stosunku do myszki

Bug wykryto obserwując działania programu. Sprawdzono za pomocą debuggera gdzie występuje błąd, a następnie wywnioskowano, że używane są dwie różne funkcje do obliczania kąta między myszą a graczem i dawały różne wyniki. Powstawał offset i dalej była możliwość obracania natomiast ray nie był skierowany na mysz. Poprawiono kod aby zamiast dwóch różnych funkcji (atan2() i acos()) korzystano z jednej.

-przy kolizji gracza z mapą nie było możliwości poruszania się w żadnym kierunku oprócz w przeciwnym do poprzedniego np. poruszając się w lewo gracz zderzał się ze ścianą, można było ruszyć się jedynie w prawej, natomiast w górę i w dół nie

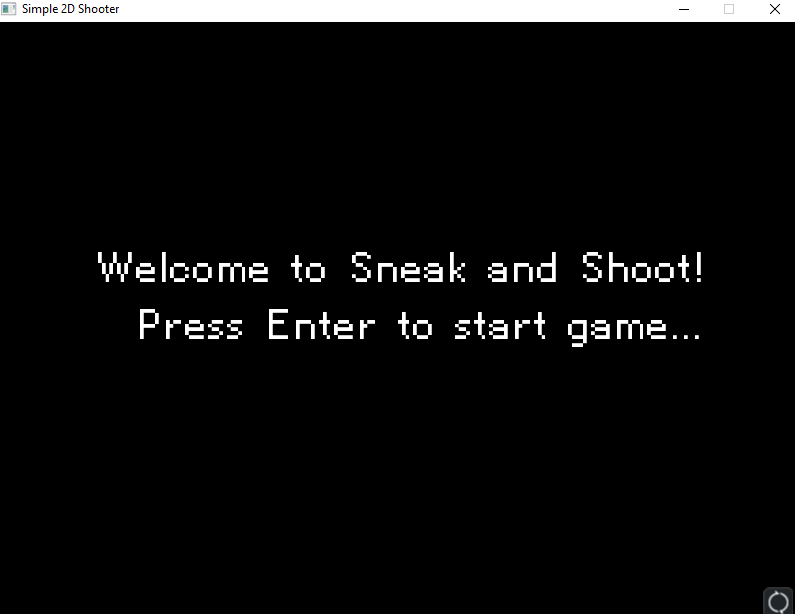
Problemem okazało się używanie funkcji ceil, natomiast nie we wszystkich przypadkach program działał poprawnie. Rozwiązaniem było użycie funkcji ceil oraz floor w zależności od przypadku. Po tej korekcji program działał poprawnie.

-przy kolizji pocisku ze ścianą następuje usunięcie pocisku, problem pojawił się gdy w vectorze przechowującym wskaźniki na pociski znajdowały się więcej niż jeden element, wtedy program przestał działać.

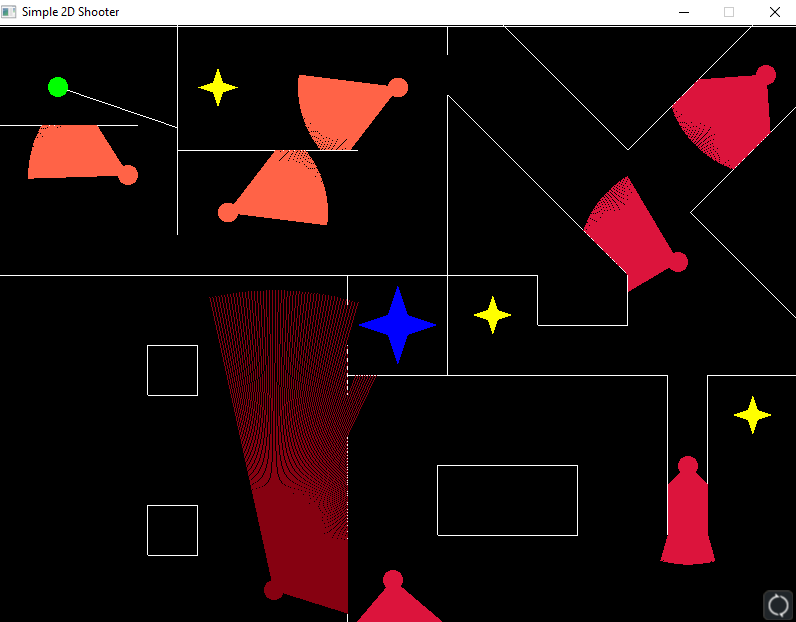
Początkowo do usuwania używano pętli for gdzie iterowano przez wszystkie elementy vectora, gdy usuwano element z vectora pętla przestawała działać co powodowało zakończenie działania programu. Do naprawy użyto funkcji lambda oraz std::remove\_if. Pomysł zaczerpnięty z serwisu stackoverflow.com. Po tej poprawce program działał poprawnie.

# **Podręcznik użytkownika**

Obsługa jest bardzo prosta ponieważ program będzie prowadził użytkownika wyświetlając komunikaty. Po uruchomieniu programu wyświetli się okno startowe. Jeżeli wyświetli się czarny ekran bez napisów, najprawdopodobniej nie udało się załadować czcionki, o czym pojawi się komunikat w konsoli. Aby rozpocząć rozgrywkę należy wcisnąć klawisz Enter. Możemy poruszać się graczem używając klawiszy WSAD, natomiast celujemy za pomocą myszki. Klikając lewy przycisk myszy oddamy strzał. Na ekranie końcowym możemy rozpocząć rozgrywkę jeszcze raz klikając klawisz R. W każdym momencie działania programu możemy go zamknąć wciskając klawisz Escape bądź zamykając okno klikając myszką krzyżyk w prawym górnym rogu.



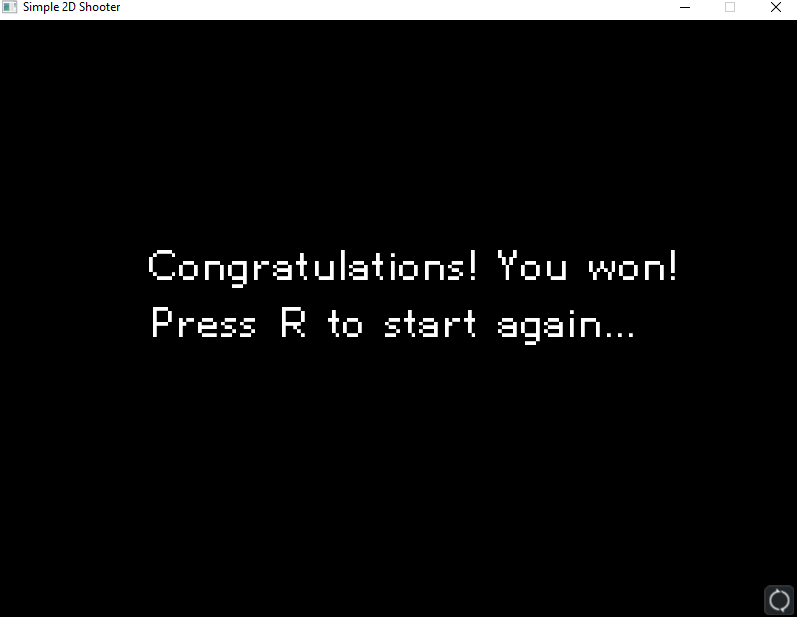
*Rys.3 Ekran Startowy*



*Rys.4 Rozgrywka*



*Rys.5 Ekran końcowy po porażce*

**

*Rys.6 Ekran końcowy po wygranej*

# **Bibliografia**

[1] McConnell Jeffrey J. : Analysis of algorithms, Jones and Bartlett Publishers, Inc

[2] <https://stackoverflow.com>

[3] <https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection>