Polynomial Fighter - dokumentacja

Projekt realizowany w ramach przedmiotu Programowanie III

Piotr Wawrzyńczyk, Bartosz Ścigała

Wydział Matematyki Stosowanej

Politechnika Śląska

16 stycznia 2018



**Spis treści**

[1 Opis programu 3](#_Toc3222)

[1.1 Opis świata przedstawionego 3](#_Toc3223)

[1.2 Interakcja z użytkownikiem 4](#_Toc3224)

[1.3 Obsługa pola tekstowego 4](#_Toc3225)

[2 Opis programu 5](#_Toc3226)

[2.1 Algorytmy 5](#_Toc3227)

[2.2 Wykorzystanie *smart pointers* 5](#_Toc3228)

[2.3 Przegląd klas oraz funkcjonalności 5](#_Toc3229)

[2.4 Zastosowane wzorce projektowe 6](#_Toc3230)

[2.5 Biblioteki 7](#_Toc3231)

[3 Testy 7](#_Toc3232)

[4 Możliwości dalszego rozwoju programu 7](#_Toc3233)

[5 Źródła multimediów 7](#_Toc3234)

[6 Wnioski końcowe 8](#_Toc3235)

# Opis programu

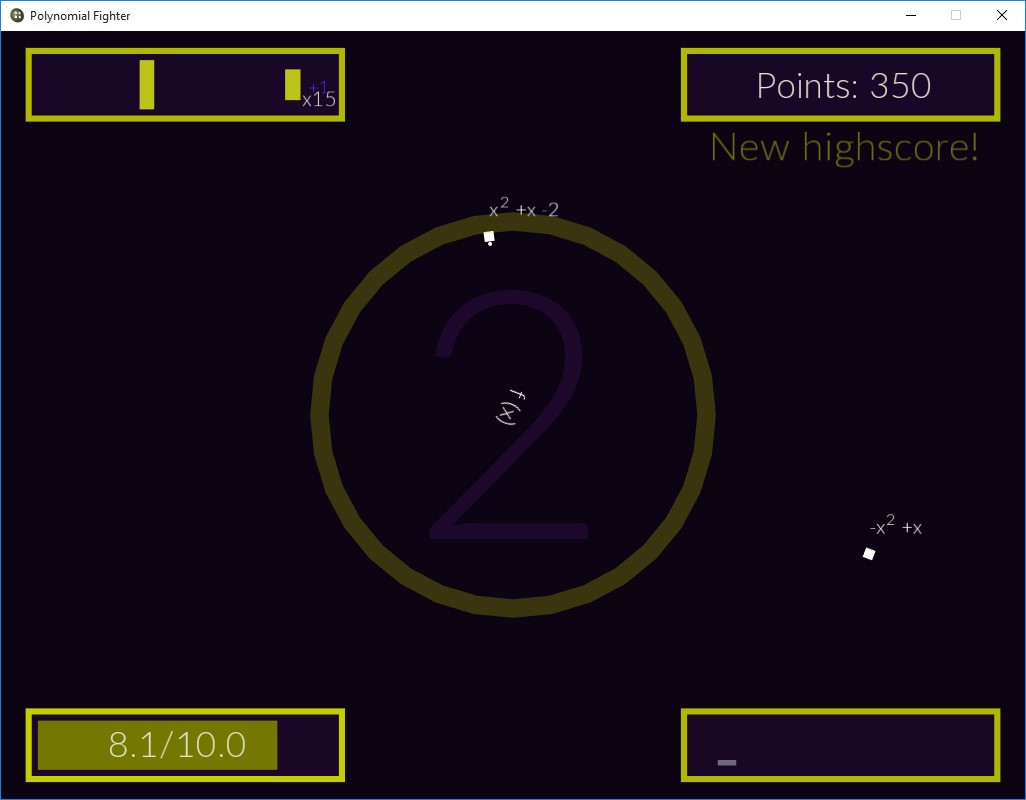
Program “Polynomial Fighter” jest edukacyjną grą komputerową, mającą na celu umożliwienie nauki płynnego rozwiązywania nieliniowych równań diofantycznych w zakresie do stopnia VII oraz współczynników poszczególnych czynników liniowych iloczynu nie większych niż 6.

## Opis świata przedstawionego



Rysunek 1: Ekran wstępny gry

Gracz jest przedstawiony jako samotna f(x), broniąca się przed nadciągającymi ze wszystkich stron[[1]](#footnote-1) wielomianami. Wielomiany, po dotarciu do okolicy pierścienia rozpoczynają ostrzał - tym cięższy i szybszy, im wyższy jest ich stopień. Punkty życia gracza są przedstawione w lewym dolnym rogu ekranu (są one tracone w momencie trafienia i odzyskiwane w momencie zniszczenia przeciwnika). Gracz musi baczyć na to, by nie zużyć całej amunicji, której licznik jest widoczny w lewym górnym rogu ekranu. Amunicja jest dodawana cyklicznie oraz - w większych ilościach - w momencie przejścia do następnego poziomu.



Rysunek 2: Przykładowy ekran gry

## Interakcja z użytkownikiem

Użytkownik może oddziaływać z grą, używając klawiatury - do poruszania się w menu używając klawiszów strzałek / WASD, do zatwierdzania klawiszy spacji lub enter, do wychodzenia / przerywania klawisza escape. Dodatkowo, w czasie gry można ją zatrzymać klawiszem ’p’.

## Obsługa pola tekstowego



Rysunek 3: Pole tekstowe z przykładowym wejściem

Pole tekstowe jest swoistym źródłem obrony wobec nadciągających wielomianów. Gdy grający wpisze właściwy pierwiastek (wyliczony z postaci wielomianu obecnego na scenie), umieszczona pośrodku planszy manifestacja gracza wyceluje w odpowiadające mu wielomiany oraz odda strzał. Gdy "życie" wielomianu - rozumiane jako jego stopień dobiegnie końca (czyli zostanie zredukowane do zera),

ten jest niszczony, a gracz otrzymuje punkty (widoczne w prawym górnym rogu okna)

# Opis programu

## Algorytmy i struktury danych

* Interpolacja liniowa - została ona wykorzystana do stworzenia płynnych przejść w menu oraz na ekranie rozgrywki.
* Tworzenie wielomianów i przejścia między ich postaciami - wielomianowe API godne osobnej biblioteki pozwoliło nam na płynne i szybkie tworzenie podstawowych mechanik gry.
* Kontenery standardowe - użycie kontenerów biblioteki standardowej STL

(takich jak vector ,array czy map) znacznie ułatwiło tworzenie gry, pozwalając nam skupić się na samej produkcji gry.

## Wykorzystanie *smart pointers*

Jedną z wielu funkcjonalności wprowadzonych w nowoczesnym C++ są tak zwane sprytne wskaźniki, odciążające programistę od dbania o zarządzanie pamięcią. Ich użycie pozwoliło na skupieniu się nad rozwiązywaniem problemów postawionych w projekcie. Dzięki nim udało nam się także utrzymać zużycie pamięci programu na stałym poziomie, mimo ciągłej dynamicznej zmianie liczby obiektów istniejących w programie.

## Przegląd klas oraz funkcjonalności

**AssetManager** Klasa singleton, reprezentująca kontener zasobów graficznych oraz dźwiękowych. Wczytane zasoby - przechowywane w *std::map* oraz identyfikowane po nazwie - są współdzielone, co pozwala na zdecydowane zmniejszenie ilości wykorzystywanej pamięci.

**FloatColor** Odpowiednik klasy sf::Color. Wykorzystywany z powodu niskiej dokładności składowych wbudowanej w bibliotekę SFML klasy przedstawiającej kolor.

**Stopwatch** Byt pomocniczy dający możliwość ustawienia stopera i odebrania zdarzenia po upływie określonego czasu.

**RandomGenerator** Klasa statyczna zapewniająca odpowiedni stopień losowości wymaganych danych, razem z opracowanymi specjalizacjami do typów (wliczono tutaj także typ sf::Color).

**Przestrzeń nazw Time** Zawiera klasę singleton Timer, uruchamianą w momencie aktywacji okna, umożliwiającą uzyskanie informacji o czasie jaki upłynął od ostatniego restartu zegara gry oraz o czasie, który upłynął od ostatniej klatki. Dane te przekazywane są w strukturze TimeData, udostępniającej metody zwracające czas z uwzględnieniem skali czasowej.

**Utility** Zbiór globalnych funkcji udostępniający wiele popularnych i niezbędnych do działania wielu innych jednostek algorytmów. Występują tu na przykład interpolacja liniowa, operacje na wektorach czy funkcje pomocnicze do obsługi rotacji.

**Debug** Klasa ze statycznymi metodami, umożliwiająca wypisywanie komunikatów na ekran konsoli. Posiada metody *PrintFormatted* oraz *PrintErrorFormatted*, powodujące wypisanie żądanego komunikatu – odpowiednio zatrzymującej i niezatrzymującej działania programu. Użyto tu szablonu funkcji, który daje możliwość wypisywania dowolnych zmiennych – pierwszym argumentem jest łańcuch znaków, dalej opcjonalnie można podać zmienne o dowolnych typach. Ich wartości zostaną wpisane do miejsc oznaczonych ‘%’ w pierwszym argumencie.

**Delegate** Nazwana na cześć podobnej funkcjonalności z języka C#, zmniejsza zapotrzebowanie na wiązanie klas poprzez opakowywanie wskaźników na funkcje. Klasa Entity przedstawia obiekt, udostępniający przeciążalne funkcje do aktualizacji (update), rysowania (draw) oraz aktywacji/dezaktywacji obiektu (set/get Enabled). Dodatkowo przechowywane są tu tag, numer ID oraz nazwa obiektu (jako łańcuchy znaków).

**EntityManager** Klasa singleton posiada kolekcję (std::vector) wskaźników na obiekty o typie Entity. W momencie gdy obiekt dziedziczący po wyżej wymienionej klasie jest tworzony (poza klasą), można zrzutować go na std::shared\_ptr<Entity>oraz dodać do kolekcji. W pętli gry uruchamiane są metoda EntityManager::update, która aktualizuje stan wszystkich aktywnych obiektów w wektorze, oraz metoda EnityManager::draw, która rysuje na ekranie aktywne obiekty. Dodatkowo udostępnione zostały metody umożliwiające wyszukiwanie obiektów po tagu, nazwie lub jego numerze ID.

**SoundManager** Kolejny przedstawiciel singletonu, ściśle współpracujący z AssetManagerem, używana do utrzymywania dźwięków w programie.

## Zastosowane wzorce projektowe

**Singleton** Dość kontrowersyjny przykład wzorca projektowego, w projekcie występował głównie w wszelkiej maści managerach, na zmianę z klasami statycznymi. Motywacją do użycia tego wzorca było to, że w przeciwieństwie do nowoczesnych języków programowania (takich jak C# lub Java), używanie klas statycznych w C++ wiążę się z dużym dodatkowym nakładem pracy i możliwością pojawienia się niespodziewanych błędów z pamięcią w przypadku nieudanej inicjalizacji statycznego obiektu, a także brak możliwości przechwytywania wyjątków przy konstruowaniu obiektów. Tak więc mimo tego, że klasy statyczne byłyby lepszym rozwiązaniem, przez ograniczenia języka C++ mogliśmy używać ich jedynie w ograniczonym zakresie (w zasadzie tylko do klas, które zawierają same funkcje, a nie muszą przechowywać wielu innych obiektów).

**Builder** Wzorzec budowniczego został użyty przy tworzeniu systemów cząsteczkowych, co znacznie uprościło ich tworzenie - zamiast jedenastoparametrowego konstruktora, można wykorzystać wartości domyślne oraz wybrać tylko te właściwości, które nas interesują.

**Łańcuch zobowiązań** Wykorzystywany do parsowania wejścia w polu tekstowym gry, pozwolił na schludne wybieranie pierwiastków na podstawie żądań użytkownika.

**Strategia** Szeroko wykorzystywany, pozwolił na uniknięcie instrukcji warunkowych na rzecz większej liczby klas. Przykładem jest obsługa klas dziedziczących po Entity.

**Odwrócenie sterowania** Wzorzec ten został wykorzystany tylko w pewnym zakresie w przypadku klasy AssetManager. W dowolnym miejscu w programie, gdzie zachodzi potrzeba użycia jakiegoś zasobu (pliku, np. graficznego), dany kod sygnalizuje tylko, jakiego zasobu potrzebuje. Nie jest wtedy ważne skąd te zasoby pochodzą, czy zostały już wczytane, czy trzeba je wczytać, czy trzeba je potem usunąć itd. Te obowiązki ciążą na szkielecie programu, dzięki czemu w podprogramach (klasach, funkcjach) mamy do czynienia z odwróceniem sterowania.

## Biblioteki

**SFML 2.4.2** Biblioteka multimedialna o szerokim możliwościach, umożliwiająca rysowanie i przetwarzanie grafiki, odtwarzanie dźwięków oraz interakcję z systemem oraz użytkownikiem.

# Testy

Program został napisany pod 64-bitową wersją systemu Windows 10 oraz 64-bitową wersją systemu Linux. Testy zostały przeprowadzone na 64-bitowych Windowsach 7, 8 oraz 10 (kompilator Visual C++ oraz MinGW 7.2) oraz na dwóch dystrybucjach Linuksa – ArchLinux oraz Ubuntu (kompilator GCC 7.2) Ważnym wymaganiem było zapewnienie na Windowsie obecności bibliotek oraz multimediów w folderze z plikiem wykonywalnym. Na Linuksie, zgodnie z zasadami dystrybuowania oprogramowania na tym systemie, biblioteka SFML 2.4 razem ze swoimi zależnościami musiała być zainstalowana w systemie. Przekazano program testerom oraz wprowadzono niezbędne poprawki, dotyczące głównie szaty graficznej oraz samej rozgrywki. Te pierwsze okazały się szczególnie cenne dla naszego doświadczenia oraz wiedzy o projektowaniu interfejsów graficznych, drugie pozwoliły uprzyjemnić oraz zbalansować rozgrywkę. Testowano również zużycie pamięci. Program początkowo zużywał zaledwie kilka megabajtów operacyjnej, jednakże wprowadzenie grafik, animacji oraz bardziej wyszukanego kroju czcionki nieco zwiększyło jego zapotrzebowanie na RAM.

Zużycia zasobów jest bardzo zadowalające. Na kilkuletnim komputerze z systemem ArchLinux, z procesorem czterordzeniowym i zewnętrzną kartą graficzną, zużycie zasobów podczas gry wynosiło:

RAM – 27 megabajty

CPU – 1-2%

GPU – 10%

Warto zaznaczyć, że gra stale działa w 120 klatkach na sekundę, więc tak niskie zużycie zasobów jest świetnym wynikiem.

# Możliwości dalszego rozwoju programu

* Dodanie muzyki (przyspieszającej w miarę rozgrywki).
* Wprowadzenie „Nowej gry+” wraz z dodatkową fabułą, przeciwnikami oraz mechanikami gry.
* Stworzenie wersji na telefony działające pod kontrolą systemu Android; interesującym konceptem okazałoby się wprowadzanie liczb za pomocą rysowania ich na ekranie.

# Źródła multimediów

* Dźwięki - opengameart.org, źródła własne
* Grafika - źródła własne

# Organizacja pracy

Jako, że nad projektem pracowały równolegle dwie osoby, odpowiednia organizacja pracy była kluczowa. Wykorzystaliśmy nasze doświadczenia z pracy zawodowej i pracowaliśmy w tzw. git workflow, tak jak ma to miejsce we wszystkich dobrych firmach zajmujących się tworzeniem oprogramowania. Używaliśmy branchy (każdy z nas pracował równolegle nad różnymi elementami, gdzie każdy taki zestaw elementów tworzył branch, np. branch w którym zostało stworzone API do wszechstronnej obsługi wielomianów), pull requestów (tutaj miało miejsce code review wykonywane przez obie osoby, a także test kodu na różnych systemach operacyjnych i kompilatorach). Do rozdzielenia pracy i katalogowania zadań do zrobienia oraz problemów do naprawienia używaliśmy issue trackera. Wszystkie te funkcjonalności zapewniło nam repozytorium kodu na GitHubie.

Dodatkowo warto podkreślić, że jedna osoba używała Windowsa (Visual Studio), a druga Linuksa (GCC), dlatego też musieliśmy pisać kod dobrej jakości zgodny ze standardem języka C++ (w naszym wypadku C++17), tak aby działał w każdym środowisku. Nie używaliśmy także żadnych API systemowych, a jedyna biblioteka jakiej użyliśmy, czyli SFML, jest w pełni wieloplatformowa. Dzięki temu cały nasz kod (bez żadnych *#ifdefów* dla poszczególnych systemów!) kompiluje się i działa poprawnie zarówno na Windowsie (Visual Studio oraz MinGW), jak i na Linuksie (GCC).

Godnym uwagi jest również fakt, iż w celu zapewnienia wysokiej jakości kodu, znalezienia ewentualnych błędów oraz zabezpieczenia programu przed błędami używaliśmy narzędzi do analizy statycznej – PVS-Studio oraz lintera wbudowanego w środowisko CLion. Analizę zużycia oraz zarządzania pamięcią przeprowadziło narzędzie Valgrind.

# Wnioski końcowe

Projekt okazał się sukcesem, zarówno w kwestia spełnienia założeń, jak i efektu końcowego. Nauka wypływająca z niego pozwoli nam lepiej spełniać swoje role jako profesjonalni deweloperzy oprogramowania.Warto tutaj podkreślić to, że projekt pomógł nam w zwiększeniu naszych umiejętności pracy zespołowej nad jedną bazą kodu, co jest prawdopodobnie najważniejszą (i najbardziej niedocenianą) umiejętnością dobrego, komercyjnego programisty.

1. Nie do końca - są oni produkowani tak, by nie nachodzili na elementy GUI. [↑](#footnote-ref-1)