1. Temat projektu

Celem projektu było odtworzenie pierwszej części z popularnej serii gier automatowych - “Mortal Kombat”, wraz z wykorzystaniem nowoczesnych technik programowania w języku C++ oraz metodyki obiektowej. Projekt przyjął żartobliwą nazwę “Patykovy Mordulec”.

2. Analiza tematu

Zadanie wymaga sprawnej realizacji wielu aspektów tworzenia gry komputerowej. Dodatkowo praca musi być w miarę możliwości równo podzielona pomiędzy 2 osoby. Najważniejsze zagadnienia:

2.1 Podstawowy silnik gry

Konieczne jest utworzenie slinika pozwalającego na wyświetlanie dwuwymiarowych grafik. Powinien on również wspierać obsługę prostej fizyki, detekcję kolizji oraz dać możliwość implementacji wymaganych nowoczesnych bibliotek języka C++.

2.2 Interfejs i nawigacja

W celu oddania stylu realizacji menu oraz interfejsu orygninalnego wzorca, należy pozostawić sterowanie nawigacją klawiaturze. Menu wyboru postaci oraz ekrany kończące pozostać mają proste i przejrzyste.

2.3 Kluczowe aspekty rozgrywki

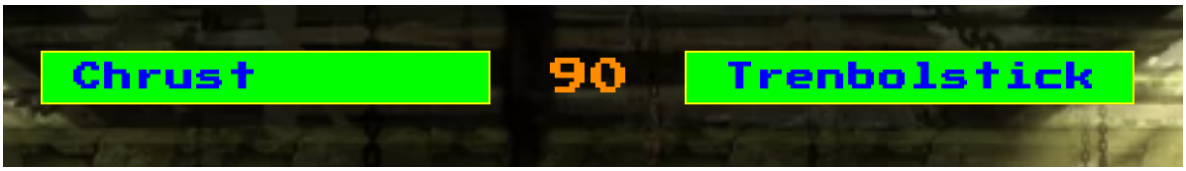
Odwzorowane mają zostać najważniejsze aspekty gry typu bijatyka, takie jak poruszanie, skakanie, wyprowadzanie różnorodych ciosów, blokowanie czy uniki. Dodatkowo gracze mają mieć możliwość wyboru postaci, które różnić będą się nie tylko wyglądem ale również zestawem ruchów oraz powiązanym z nim “stylem walki”.

2.4 Cykl trwania meczu

Każdy mecz podzielony zostanie na niezależne rundy. Po rozpoczęniu rozgrywki czas do zakończenia rundy pozostanie ograniczony. Rudna trwać będzie do momentu, aż któremuś z graczy skończą się punkty zdrowia, lub do upłynięcia czasu. Mecz kończy się w momencie wygrania przez jednego z graczy 2 rund.

2.5 Elementy GUI

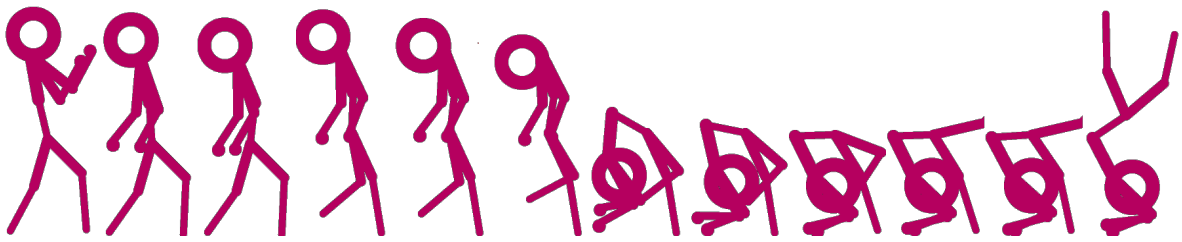
Na ekranie gry poza samymi postaciami znajdować będą się ulokowane w stałych miejscach elementy Graficznego Interfejsu Użytkownika. Będą to między innymi paski, reprezentujące ilość posiadanych punktów zdrowia każdego z graczy oraz licznik czasu rundy. Ich wygląd również wzorowany będzie na oryginalnej inspiracji.



Rys. 1: Finalna wersja głównych elementów GUI

2.6 Treści i zawartość

Poza typowo technicznymi aspektami, projekt zakłada dodanie do napisanej gry podstawowych zawartości. W celu oddania oryginalnej oprawy graficznej rysowane elementy, takie jak grafiki postaci lub areny wyglądem będą przypominały grę retro. Dodatkowo utworzone będą oryginalne postacie, wraz z pasujacymi do ich wyglądu zestawami ruchów. Same postacie będą musiały przejść przez proces optymalizacji oraz balansowania, w celu zapewnienia sprawiedliwej i zajmującej rozgrywki.



Rys. 2: Klatki animacji ataku jednej z postaci

2.7 Optymalizacja

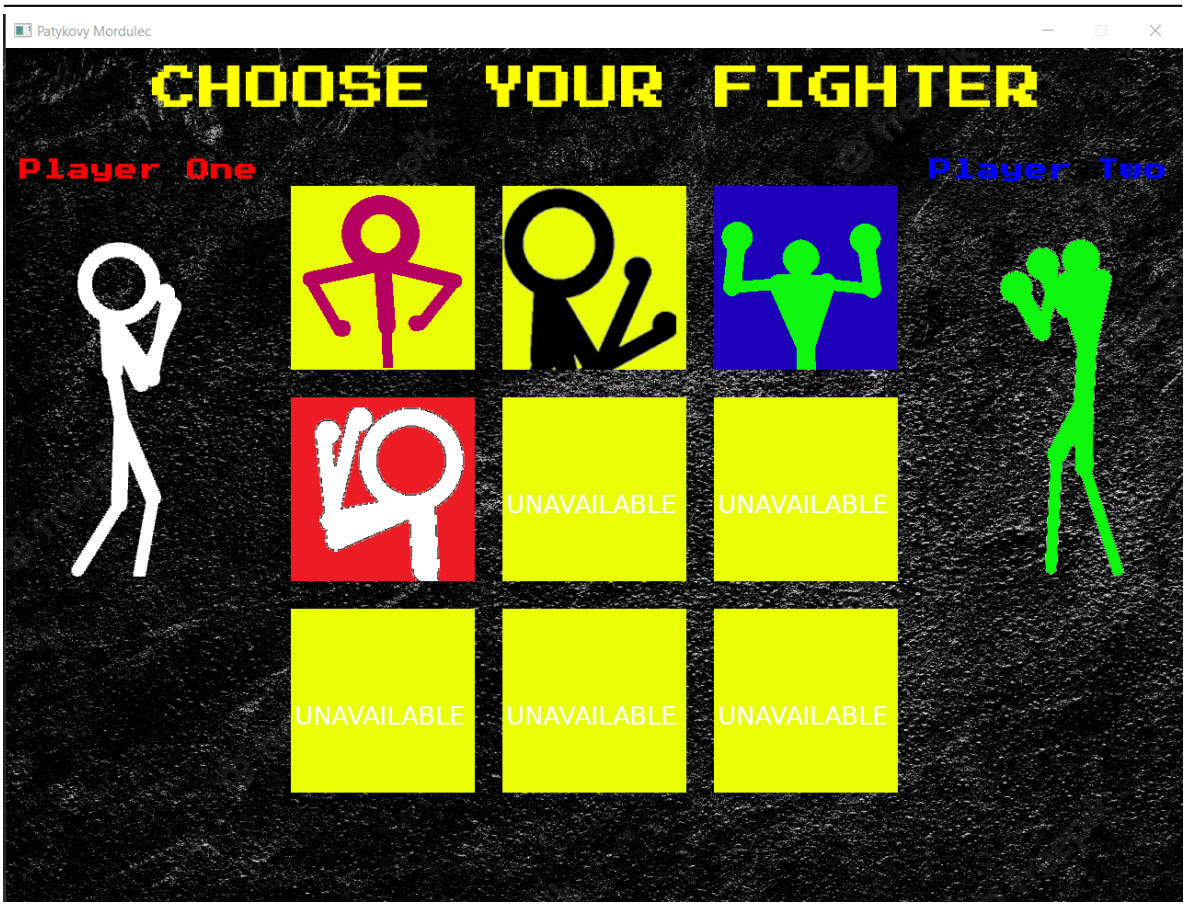
Gry gatunku bijatyka mocno opierają się na szybkich reakjcach graczy. Z tego powodu płynna i responsywna rozgrywka jest wyjątkowo istotna. Wymagane jest, aby silnik posostawiał wiele miejsca na optymalizację. Ważne jest unikanie czasochłonnych operacji oraz sprawna identyfikacja punków krytycznych.

2.8 Biblioteki zewnętrzne

W celu realizacji podstawowych elementów silnika oraz rozgrywki zdecydowano się wykorzystać bibliotekę Simple and Fast Multimedia Library. Pozwala ona na proste wprowadzanie wielu istotnych mechanik, takich jak wczytywanie i wyświetlanie grafik, operacje związane z liczeniem czasu lub detekcję kolizji.

3. Specyfikacja zewnętrzna

Program można uruchomić z linii poleceń, lub klikając dwukrotnie na plik wykonywalny programu. Po uruchomieniu, wyświetla się ekran powitalny, który po 5 sekundach przechodzi do ekranu wyboru postaci - menu.



Rys. 3: Ekran wyboru postaci

Gracze mogą wybierać spośród 4 unikalnych postaci, za pomocą klawiszy *WSAD* oraz *IKJL*. Portrety aktualnie wybranych postaci posiadają odpowiednie kolory tła, dla gracza pierwszego czerwone, a dla gracza drugiego - niebieskie. Oprócz tego, po lewej i prawej stronie ekranu wyświetlana jest animacja, w której biorą udział aktualnie wybrane postacie. Aby zakończyć etap wyboru postaci i przejść do rozgrywki, gracz pierwszy powinien wcisnąć i przytrzymać klawisz *TAB* a gracz drugi, klawisz *ENTER*.

4. Specyfikacja wewnętrzna

4.1 Diagram klas

Diagram klas, zgodny ze standardami UML, wraz z wydzielonymi odpowidzialnościami każdego z autorów, znajduje się w załączniku.

4.2 Wykorzystane nowoczesne biblioteki języka C++

4.2.1 Biblioteki menagera scen, wczytywania i GUI - Wojciech Ptaś

lorem

4.2.2 Biblioteki silnika, animatora oraz mechanik - Jan Kocurek

Korzystając z funkcjonalności *modules* standardu C++20 utworzono moduł `async_functions.ixx`. Zawiera on zestawy operacji, których wykonywanie jest niezależne od innych elmentów kodu. Dzięki temu można wykonywać je asynchronicznie, korzystając z funkcjonalności biblioteki standardowej `std::async`.

```
export module async_functions;

export void async_read_input(GameEngine* g);
export void async_move_players(std::shared_ptr<Player> p1, std::shared_ptr<Player>
↪  p2);
export void async_recovery(std::shared_ptr<Player> p);
export void async_animation(std::shared_ptr<Player> p);
export const bool update_view(GameEngine* g, std::shared_ptr<Player> p1,
↪  std::shared_ptr<Player> p2, std::shared_ptr<sf::View> view, const float&
↪  dist_between_players);
export void init_gameobject_variables(GameObject* gameObject);

module :private;
//ciała funkcji
```

Fragment kodu modułu

```
std::thread animation_th_p1(async_animation, this->player1);
std::thread animation_th_p2(async_animation, this->player2);

animation_th_p1.join();
animation_th_p2.join();
```

Przykład zastosowania, fragment `GameEngine.cpp`

Dodatkowo, dwie z funkcji modułu wykorzystuje funkcjonalość `std::semaphore`. Elementy kalkulacji ruchu postaci oraz odczytywania wprowadzanej przez gracza kombinacji klawiszy wykonują się jednocześnie. W momencie, gdy odczyt jest gotowy, funkcja `async_read_input` zwalnia semafor, na który oczekuje funkcja `async_move_players`.

```
void async_read_input(GameEngine* g) {
    g->updateInput();
    prepare.release();
}

void async_move_players(std::shared_ptr<Player> p1, std::shared_ptr<Player> p2) {
    bool can_p1_move = p1->canMove();
    bool can_p2_move = p2->canMove();
    prepare.acquire();

    if (can_p1_move) {
        p1->duck();
        p1->move();
        p1->jump();
    }
    if (can_p2_move) {
```

```
        p2->duck();
        p2->move();
        p2->jump();
    }
}
```

Omawiany fragment modułu

Kontener tekstur każdej z postaci przed wczytaniem inicializowany jest wartościami `nullptr`. Jeśli odczyt którejś z funkcji nie powiedzie się, w wielu miejscach zgłoszony zostać może błąd dostępu. W celu weryfikacji poprawnego odczytania tekstur wykrozystano funkcjonalość `std::ranges`.

```
auto didLoad = [](const std::shared_ptr<sf::Texture> pointer) { return pointer ==
↪  nullptr; };
for (const auto& val : std::views::values(this->playerTextures) |
↪  std::views::filter(didLoad)) {
        std::cout << "PLAYER::initTexturesMap() CRITCAL ERROR: nullptr in map!\n";
}
```

Zastosowanie biblioteki `ranges`

5. Testowanie

Program był wielokrotnie uruchamiany i rozgrywany różnymi dostępnymi postaciami. # 6. Uwagi i wnioski

6.1 Praca zespołowa i organizacja

Elementem który pozowlił utrzymać wysoką produktywność oraz uniknąć wielu kłopotów podczas późniejszych faz realizacji projektu było przykładne i konkretne podzielenie się obowiązkami oraz ustalenie wspólnej wizji produtku końcowego w pierwszej fazie pracy zadaniem. Praca została rozłożona w taki sposób, aby obydwie osoby pracujące nad projektem nie przeszkadzały sobie nawzajem. Tworzone komponenty pozostały mało zależne od siebie, a interfejsy kompatybilne. W ten sposób mogły być tworzone, testowane i modyfikowane jednocześnie bez wzajemnego utrudniania pracy. Wyjątkowo przydatne okazało się korzystanie z systemu kontroli wersji GIT. Umożliwił on bezproblemową współpracę nad kodem oraz pomógł lepiej kontrolować historię pisanego kodu.

6.2 Kłopotliwe aspekty produkcji gier

6.3 Nowoczesne biblioteki języka C++