Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka techniczna (ITE)**
Specjalność: **Inżynieria systemów informatycznych (INS)**

PRACA DYPLOMOWA INŻYNIERSKA

Projekt gry z gatunku bijatyk 2D umożliwiającej prowadzenie rozgrywek online

A project of a 2D fighting game that allows online play

Jan Kucharski

Opiekun pracy
dr inż. Tomasz Kubik

Słowa kluczowe: aplikacja desktopowa, gra z gatunku bijatyk 2D, interakcja z użytkownikiem, tryb offline, tryb online

WROCŁAW, 2024

Streszczenie

W pracy dokładnie opisano proces powstawania tytułowej gry. W pierwszej części pracy skupiono się na analizie założeń projektowych, definiując główne funkcje aplikacji, jej interfejs użytkownika oraz unikalne cechy, które wyróżniają ją na tle innych gier tego gatunku. Następnie szczegółowo omówiono etap implementacji, w tym wybór i wykorzystanie technologii Java i frameworku libGDX. W ramach tego etapu rozważano różnorodne aspekty techniczne, od opracowania mechanik rozgrywki, systemu kolizji, animacji postaci, po zarządzanie danymi i interakcje sieciowe. Kolejna część pracy poświęcona jest testowaniu gry, zarówno w aspekcie funkcjonalnym, jak i w zakresie stabilności i wydajności. Przeprowadzone testy miały na celu zapewnienie, że gra spełnia wszystkie postawione założenia. Ostatni rozdział pracy zawiera podsumowanie całego procesu tworzenia gry, wyciąganie wniosków i zaleceń na przyszłość. Praca ta stanowi źródło wiedzy na temat procesu projektowania, implementacji i testowania gier komputerowych, z naciskiem na gry z kategorii dwuwymiarowych bijatyk i opisuje wyzwania związane z tworzeniem gier online.

Słowa kluczowe: aplikacja desktopowa, gra z gatunku bijatyk 2D, interakcja z użytkownikiem, tryb offline, tryb online

Abstract

The thesis provides details of the process of creating the title game. It begins with an analysis of the project's foundational concepts, defining the application's main functions, user interface, and unique features that set it apart from other games in its genre. The implementation phase, including the selection and application of Java technology and the libGDX framework, is thoroughly discussed. This part of the thesis considers various technical aspects, from game mechanics and collision systems to character animation, data management, and network interactions. Subsequent sections are dedicated to game testing, both functional and in terms of stability and performance, ensuring that the game meets all set objectives. The final chapter summarizes the entire game development process, drawing conclusions and recommendations for future work. This thesis serves as a knowledge resource on the design, implementation, and testing process of computer games, emphasizing 2D fighting games and the challenges of creating online games.

Keywords: desktop application, 2D fighting game, user interaction, offline mode, online mode

Spis treści

Spis rysunków

Spis tabel

Spis listingów

Skróty

2D – dwuwymiarowy (ang. *two-dimensional*),
3D – trójwymiarowy (ang. *three-dimensional*),
FGC – społeczność graczy gier z gatunku bijatyk (ang. *Fighting Game Community*),
IDE – zintegrowane środowisko programistyczne (ang. *Integrated Development Environment*),
IP – protokół internetowy (ang. *Internet Protocol*),
ISP – dostawca usług internetowych (ang. *Internet Service Provider*),
UPnP – uniwersalna funkcja Plug and Play (ang. *Universal Plug and Play*).

Rozdział 1

Wstęp

1.1. Wprowadzenie

Świat gier komputerowych, w szczególności gier prowadzonych w trybie *on-line*, budowany jest zwykle w oparciu o różnego rodzaju mechanizmy rywalizacji. Dzięki tym mechanizmom samo granie przestaje mieć formę beztroskiej rozrywki, a staje się czymś więcej – sposobem na sprawdzenie własnych możliwości, testem osiągniętej sprawności, okazją do zbudowania pozycji itp. Od wprowadzonych zasad oraz zakresu obsługiwanych możliwości interakcji między graczami zależy, czy grono użytkowników danej gry powiększać się będzie o kolejne grupy fanów, czy wręcz przeciwnie, dana gra nie zdobędzie żadnej popularności. Niespecjalnie to nawet dziwi, gdyż rywalizacja jest uważana za część ludzkiej egzystencji, zaś uczestnictwo w cyfrowych konfliktach to tylko realizacja głęboko zakorzenionej potrzeby.

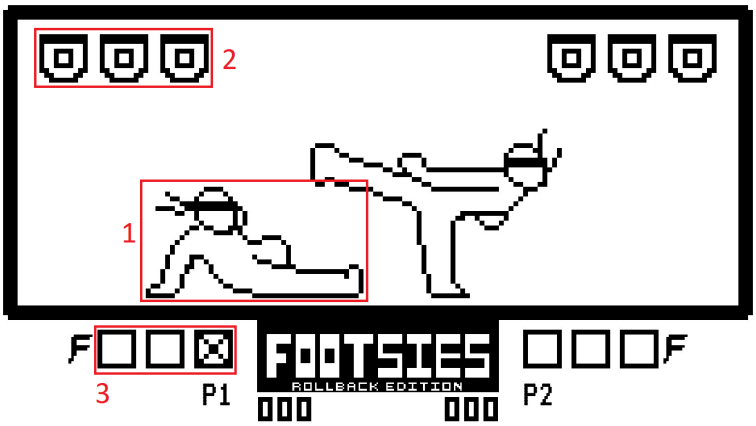
Wydawaniu gier komputerowych niezmiennie towarzyszy tworzenie się wokół nich lokalnych społeczności, zrzeszających osoby o podobnych zainteresowaniach. Przykładem takiej społeczności jest, ciesząca się zasłużoną renomą, społeczność FGC (ang. *Fighting Game Community*). Jej członkowie, włącznie z autorem niniejszej pracy, rywalizują ze sobą, dzielą się wskazówkami, transmitują swoje rozgrywki oraz, co równie istotne, budują nowe relacje przyjacielskie. Dla wielu z nich udział w turniejach to nie tylko okazja do rywalizacji, ale także pretekst do spotkania ze starymi znajomymi dzielącymi tę samą pasję.

Pojawiający się w angielskiej nazwie grupy termin *Fighting Games* wskazuje na pewien gatunek gier, wokół których skupia się ta społeczność. W tłumaczeniu na język polski tego typu gry określa się mianem „bijatyk". W „bijatykach" rywalizacja pomiędzy graczami odbywa się na wzór rywalizacji w sztukach walki – gracze wykonują ruchy awatarami, zadając sobie ciosy, blokując ataki i stosując uniki. Cała rozgrywka obywa się w wirtualnym świecie o różnej złożoności, od dwuwymiarowej sceny, poprzez rozbudowane mapy, aż po trójwymiarowe środowiska renderowane z dokładnością bliską filmowanej rzeczywistości.

Wspomniane kulturowe otoczenia bijatyk, jak również budowany od dziecięcych lat sentyment do tego typu gier, okazały się kluczowymi czynnikami do zdefiniowania i podjęcia przez autora tematu niniejszej pracy dyplomowej. Jego realizacja, ujmując to w skrócie, polegać ma na stworzeniu sieciowej gry z gatunku bijatyk, oferującej prostą, dwuwymiarową wizualizację sceny oraz proste sterowanie.

Dobrym przykładem ilustrującym działanie gier z gatunku bijatyk jest gra FOOTSIES: `https://store.steampowered.com/app/1344740/FOOTSIES_Rollback_Edition/`.

Na rysunku 1.1 pokazano jeden z głównych widoków prezentowanych uczestnikom gry, zaznaczając na nim główne jego elementy. Widok ten ma ułatwić zrozumienie, o czym będzie niniejsza praca.



Rys. 1.1: Zrzut ekranu z gry FOOTSIES z zaznaczonymi głównymi elementami gry: 1 - postać którą steruje gracz, 2 - licznik punktów bloku, 3 - licznik wygranych rund

Jak można się domyślać, gracz kontroluje reprezentującą go postać (awatara), rywalizując z innym graczem (z jego awatarem) w kolejnych rundach. Celem danej rundy jest powalenie przeciwnika. Kto tego dokona, wygrywa daną rundę. Wygranie trzech rund oznacza zwycięstwo w całej grze (i jej zakończenie).

Gracze mają możliwość wykonywania różnych ruchów. Mogą na przykład przemieszczać awatara po planszy, poruszając nim w stronę przeciwnika lub cofając go. Warto zaznaczyć, że cofanie pozwala zablokować nadchodzące ataki. Odbywa się to jednak kosztem utraty punktów bloku. Wyzerowanie konta punktów bloku sprawia, że dana postać awatara staje się bezbronna.

Jeśli chodzi o ataki, to wyróżnia się ich dwa główne: ataki normalne i ataki kończące. Atak normalny jest stosunkowo szybki, co może okazać się przydatne, jako że podczas animacji jakiegokolwiek ataku nie możemy blokować. Służy on głównie do testowania przeciwnika i zmniejszania jego punktów bloku. Każdy normalny atak ma kontynuację w postaci ataku kończącego. Jeśli uda się trafić przeciwnika atakiem kończącym, przeciwnik przewraca się, a to oznacza zwycięstwo w rundzie. Ataki kończące są znacznie wolniejsze i odsłaniają postać atakującego na odpowiedź przeciwnika (jest wystarczająco dużo czasu, aby przeciwnik zadał kontrujący atak kończący). Do tego dochodzą dodatkowe właściwości poszczególnych ciosów, jak np. niewrażliwość na ciosy przeciwnika podczas animacji.

Cała mechanika gry pozwala na wdrożenie interesujących strategii. Mogą one obejmować próby zmylenia przeciwnika i skłonienia go do podjęcia złego ruchu, zazwyczaj poprzez stworzenie sytuacji, która wskazuje na zamiar zrobienia jakiegoś konkretnego ruchu, gdy faktycznie w planach atakującego jest coś innego. Przykładowo, *whiff punishing* to jeden z elementów taktyki polegającej na kontrolowaniu odległości w trakcie walki. Konkretnie oznacza on skaranie przeciwnika za chybienie podczas swojego ataku. Idea wykorzystująca tę taktykę może polegać na zbliżeniu się do przeciwnika w taki sposób, aby zachęcić go do podjęcia ataku, a następnie szybko się cofnąć, obserwując, jak przeciwnik popełnia błąd i skontrować go.

1.2. Cel i zakres pracy

Celem niniejszego projektu jest opracowanie oraz pełna implementacja desktopowej, dwuwymiarowej gry zręcznościowej, zaliczającej się do kategorii bijatyk. Gra ta ma umożliwiać rywalizację pomiędzy graczami w trybie online, przy wykorzystaniu uproszczonych rozwiązań w zakresie sterowania, grafiki i mechaniki walki.

Realizacja pracy rozpocznie się od zdefiniowania wymagań. Ważne na tym etapie będzie zwrócenie uwagi na możliwe ścieżki implementacji gry. Następnie, po dokonaniu wyboru od-

powiedniej technologii i konfiguracji środowiska programistycznego, nastąpi etap projektowania i implementacji. W ramach tej części pracy zostanie stworzony interfejs użytkownika oraz opracowana mechanika rozgrywki. Kolejnym krokiem będzie napisanie kodu źródłowego oraz stworzenie grafik. Cały projekt zostanie poddany szeregowi testów celem zapewnienia jego poprawnej działalności. Ostatecznie zostanie przygotowana pełna dokumentacja projektu, włączając w nią instrukcję obsługi.

1.3. Układ pracy

W rozdziale pierwszym przedstawiono wprowadzenie do tematu gier z gatunku bijatyk, oraz zakres, cel i układ pracy. W rozdziale drugim opisano założenia projektowe. W kolejnym, trzecim rozdziale, przedstawiono szczegóły implementacji, łącznie z opisem fragmentów kodu źródłowego. W rozdziale czwartym zwrócono uwagę na testy oraz uzyskane wyniki. Ostatni, piąty rozdział, przeznaczono na podsumowanie. Pracy towarzyszy wykaz literatury oraz dwa dodatki.

Rozdział 2

Założenia projektowe

2.1. Wymagania funkcjonalne

Sposób zachowania oraz oczekiwane funkcje gry opisano w poniższych podrozdziałach.

2.1.1. Reguły i mechanika gry

Gra stworzona w ramach projektu będzie nazywała się *Sunset Showdown*. Charakterystyczne dla gry będzie ujednolicenie postaci sterowanymi przez graczy. Każda z postaci będzie miała 3 punkty zdrowia, a zwycięstwo w rundzie będzie osiągane poprzez zredukowanie zdrowia przeciwnika do zera. W przypadku trzykrotnego powodzenia tego procesu, gracz wygrywa całą grę.

Postacie będą miały możliwość zadawania ciosów na trzech różnych wysokościach: *high* (cios wysoki), *mid* (cios wyprowadzony w środkową część ciała) oraz *low* (cios niski). Wysokość ciosu wpływa na to, w jakim stanie można go zablokować lub uniknąć. Cios *high* można zablokować stojąc, uniknąć zaś kucając. Cios *mid* można zablokować w pozycji stojącej, jednak nie chroni przed nim kucanie. Natomiast ciosem *low* można trafić stojącego przeciwnika, przy czym może on go zablokować kucając. Dodatkowo, każdy cios będzie zadawany z inną szybkością. Atak *high* będzie najszybszy, *mid* zajmie środkową pozycję, natomiast cios *low* będzie najwolniejszy.

Każdy atak zadaje określoną ilość obrażeń (*high* – 3 obr., *mid* – 2 obr., *low* – 1 obr.). Dodatkowo każdy cios będzie powodował inną animację u przeciwnika na bloku (tzw. *blockstun*), od której będzie zależało, jak będzie wyglądała sytuacja po zablokowanym ataku. Po zablokowaniu ataku *high*, atakujący znajduje się w lepszej pozycji, gdyż *blockstun* trwa na tyle długo, że atakujący będzie mógł szybciej odzyskać kontrolę nad postacią. W przypadku ataku *mid*, sytuacja odwraca się, a po zablokowaniu obrońca znajduje się w lepszej pozycji. Sytuacja po zablokowaniu ciosu *low* jest szczególnie dynamiczna, ponieważ blokujący ma wystarczająco dużo czasu, aby zareagować ciosem *high*, co równa się z wygraniem rundy (tzw. *block punishment*).

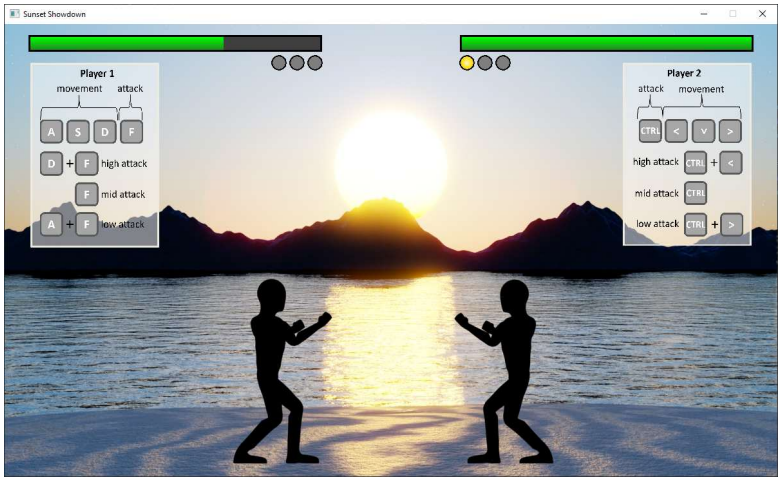
Wszystkie kluczowe informacje na temat ciosów postaci znajdują się w tabeli 2.1. Wartości szybkości ruchów i sytuacji zarówno na trafieniu i na bloku są podawane w „klatkach", które są równoznaczne z 1/60 sekundy (np. jeżeli sytuacja na trafieniu wynosi +2 oznacza to że postać atakującego po trafieniu będzie mogła wrócić do kontroli nad swoją postacią o 2/60 sekundy szybciej niż przeciwnik).

Tab. 2.1: Atrybuty ataków dostępnych dla postaci

Wysokość ataku	Szybkość ataku	Obrażenia	Sytuacja na bloku	Sytuacja na trafieniu
`high`	15	3	+1	nie dotyczy
`mid`	18	2	-2	+2
`low`	21	1	-15	+2

2.1.2. Interfejs użytkownika

Na rysunku 2.1 przedstawiono wygląd interfejsu użytkownika gry *Sunset Showdown*. Postacie graczy zobrazowane są na nim jako sylwetki w jednolitym, czarnym kolorze. W górnej części ekranu znajdują się zielone paski zdrowia dla każdego z graczy. Zmniejszają się one w miarę otrzymywania obrażeń przez postacie. Poniżej pasków zdrowia znajdują się ikony w kształcie kółek. Jeżeli są zapalone na żółto, to wskazują na wygrane rundy.



Rys. 2.1: Widok intefejsu użytkownika gry *Sunset Showdown*

Po bokach ekranu umiejscowione są schematy sterowania dla każdego z graczy, przedstawiające przypisane klawisze i ich funkcje w grze. Dla Gracza 1, po lewej stronie, przyciski 'A', 'S', 'D', i 'F' odpowiadają za ruchy i podstawowe ataki, z dodatkowymi kombinacjami klawiszy dla ataków wysokich, średnich i niskich. Analogicznie, Gracz 2, po prawej stronie ekranu, korzysta z klawiszy strzałek oraz 'CTRL' do sterowania swoją postacią i wykonania ataków.

2.1.3. Tryby gry

W *Sunset Showdown* dostępne będą dwa podstawowe tryby gry. Tryb offline umożliwi dwóm graczom wspólną rozgrywkę na jednym komputerze przy użyciu tej samej klawiatury. Oprócz tego gra będzie oferować tryb online, który pozwoli graczom na połączenie się przez Internet z osobnych komputerów. W tym trybie gracze będą mogli tworzyć pokoje gry, do których dostęp jest możliwy po wprowadzeniu specjalnego kodu pokoju. Funkcje te będą dostępne w menu głównym gry.

2.2. Wymagania niefunkcjonalne

W niniejszym podrozdziale zebrano wymagania niefunkcjonalne, w tym wymagania odnośnie wykorzystanych narzędzi i technologii oraz wymagania systemowe.

2.2.1. Narzędzia i technologie

Projekt gry zostanie zrealizowany w języku Java [15], z wykorzystaniem Java Development Kit (JDK) wersja 17. Dodatkowo wykorzystany zostanie framework LibGDX [16]. Framework ten będzie odpowiedzialny za szereg funkcji, takich jak ładowanie zasobów, wyświetlanie sprite'ów, odtwarzanie dźwięków, a także zarządzanie główną pętlą gry i interfejsem użytkownika w systemie Windows.

Do serializacji danych zastosowana zostanie biblioteka Kryo [4], natomiast aspekty sieciowe obsłuży Kryonet [5] wraz z biblioteką weupnp [1]. Ponadto, do stworzenia pomocniczego programu przeznaczonego do generowania plików JSON, które szczegółowo opisują ruchy postaci, wykorzystam bibliotekę JavaFX [13].

Proces tworzenia modeli 3D i ich animacji, zarówno dla postaci, jak i tła, zostanie wykonany przy użyciu programu Blender [3]. Natomiast do kreowania spritesheet'ów wybór padł na ImageMagick [9]. Ostatnią częścią projektu będzie udźwiękowienie gry, do czego posłużą darmowe dźwięki pobrane ze strony pixabay [14].

2.2.2. Wymagania systemowe

Aby uruchomić grę, konieczne jest posiadanie komputera z systemem operacyjnym Windows (najlepiej Windows 10). Do wykorzystania trybu online niezbędne jest połączenie z internetem. W przypadku pełnienia roli hosta (założyciela pokoju dołączanego przez innych graczy), wymagana jest odpowiednia konfiguracja sieciowa: komputer musi być połączony z routerem, który z kolei powinien być bezpośrednio podłączony do dostawcy usług internetowych ISP (ang. *Interner Service Provider*). Istotna jest również aktywna funkcja UPnP (ang. *Universal Plug and Play*) na routerze, umożliwiająca automatyczne przekierowywanie portów. Minimalne wymagania sprzętowe:

- procesor – AMD Ryzen 3 2200U,
- karta graficzna – AMD Radeon Vega3 Mobile Graphics,
- pamięć RAM: – 8 GB,
- wolne miejsce na dysku – 200 MB.

2.2.3. Środowisko programistyczne

W przypadku decyzji o uruchomieniu aplikacji poprzez skompilowanie dostępnego w publicznym repozytorium kodu źródłowego, konieczne będzie spełnienie dodatkowych wymagań infrastrukturalnych. Do nich należą:

- zintegrowane środowisko programistyczne IDE (ang. *Integrated Development Environment*) – zalecane Intellij IDEA [10];
- system zarządzania projektami i zależnościami – Gradle,
- system wersjonowania kodu – platformy GitHub [8] oraz program git [7].

Po otwarciu projektu w IDE, pozostałe zależności powinny zostać automatycznie pobrane. Ważne jest również, aby upewnić się, że folder roboczy uruchamianej aplikacji jest folderem assets znajdującym się bezpośrednio w głównym katalogu projektu.

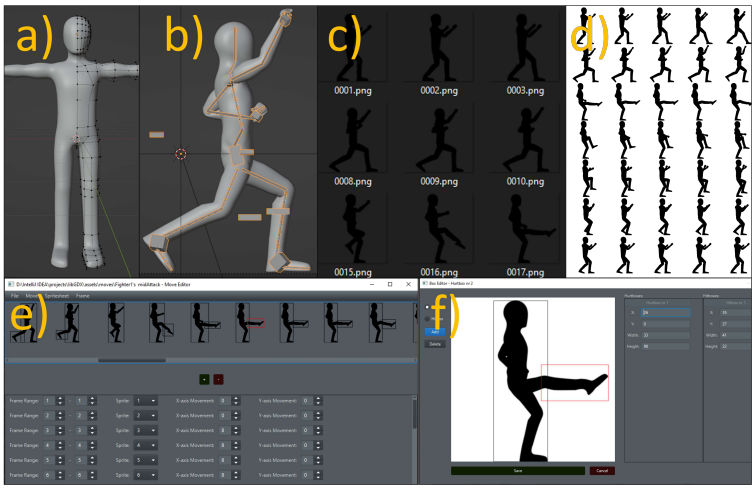
Rozdział 3

Implementacja

W niniejszym rozdziale przedstawiono szczegółową analizę implementacji dwuwymiarowej gry zręcznościowej z gatunku bijatyk. Omówienie obejmuje kluczowe aspekty techniczne projektu, takie jak tworzenie animacji, struktura klas, wzorce projektowe oraz implementacja mechanik gry, zarówno w trybie offline, jak i online. Szczególny nacisk położono na takie elementy, jak zarządzanie postaciami, detekcja kolizji, obsługa sieci oraz interfejs użytkownika, które razem stanowią rdzeń funkcjonalności i grywalności projektu.

3.1. Tworzenie animacji

Na rysunku 3.1 pokazano proces tworzenia animacji. Na początku postać została zamodelowana w 3D. Użyto do tego narzędzia Blender. Następnie przeprowadzono *rigging*, czyli proces dodawania szkieletu do modelu 3D, który pozwala na animowanie poszczególnych części ciała. Po ustawieniu postaci w pożądanych pozach, każda klatka animacji była renderowana i zapisywana jako seria oddzielnych plików graficznych (*sprite'ów*). Te obrazy stanowiły podstawę dla każdego ruchu postaci w grze. Następnie używając narzędzia ImageMagick *sprite'y* zostały połączone w jeden większy obraz zwany *spritesheet'em*. Używając stworzonego przez autora pracy, na potrzeby tego projektu, narzędzia do edycji animacji Move Editor, stworzono wpisy do każdej klatki, opisujące między innymi ruch postaci. W Move Editorze określano również *hurtbox'y* i *hitbox'y* dla każdej klatki animacji. *Hurtbox* to obszar, w którym postać może otrzymać

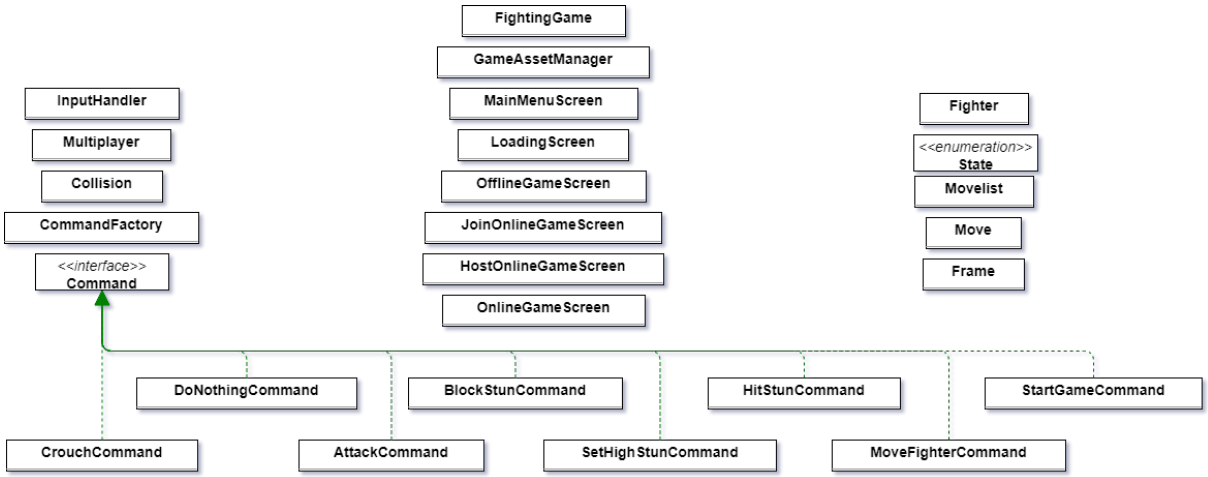


Rys. 3.1: Proces tworzenia animacji: a) modelowanie, b) *rigging*, c) wygenerowanie *sprite'ów*, d) stworzenie *spritesheet'a*, e) edycja klatek, f) definicja *hurtbox'ów* i *hitbox'ów*

obrażenia, natomiast *hitbox* to obszar, w którym postać zadaje obrażenia. Ta funkcjonalność jest kluczowa dla mechaniki gry, ponieważ decyduje o tym, kiedy ataki postaci trafiają lub są blokowane. Po zakończeniu edycji *spritesheet* wraz z opisem klatek w formacie json zostały zapisane w odpowiedniej strukturze plików, aby mogły być bezpośrednio wykorzystane w grze. Oprócz animacji samych postaci została również wykonana animacja zachodzącego słońca, która następnie została wykorzystana do stworzenia dynamicznego tła wyświetlającego się podczas rozgrywki.

3.2. Struktura klas

Na rysunku 3.2 przedstawiono główne klasy stworzone na potrzeby gry. W poniższych podsekcjach zamieszczono krótki opis każdej z nich.



Rys. 3.2: Struktura zaprojektowanych klas

3.2.1. Główna struktura gry

`FightingGame` służy jako centrum zarządzania głównymi aspektami gry, takimi jak inicjalizacja, cykl życia gry oraz przełączanie pomiędzy różnymi ekranami (np. menu, gra itp.). W procesie inicjalizacji wykorzystuje klasę `GameAssetManager`, która odpowiada za wczytywanie i zarządzanie zasobami gry, takimi jak tekstury, dźwięki i inne elementy multimedialne.

`MainMenuScreen`, `LoadingScreen`, `OfflineGameScreen`, `JoinOnlineGameScreen`, `HostOnlineGameScreen` oraz `OnlineGameScreen` są klasami reprezentującymi różne ekrany w grze, każdy z nich implementuje interfejs `Screen` dostarczony przez framework `libGDX`. Zapewniają one różnorodne interfejsy uzytkownika i są odpowiedzialne za prezentację odpowiedniego widoku w zależności od stanu gry.

3.2.2. Mechanika rozgrywki

Klasa `Fighter` reprezentuje postać walczącą. Przechowuje ona wartości takie jak zdrowie, pozycję, *hurtbox'y* i *hitbox'y* postaci oraz stan w którym aktualnie się znajduje. Ten stan reprezentowany jest przez typ wyliczeniowy `State` i obejmuje takie sytuacje jak „chodzenie do przodu", „cofanie się", „kucanie" jak i wykonywanie różnych ataków albo otrzymywanie ich na bloku lub bez niego. Na podstawie tej klasy często wykonuje się logika reszty programu.

Awatar, którym steruje gracz, posiada zestaw ruchów zawarty w klasie `Movelist`. Ta klasa przetrzymuje poszczególne ruchy o klasie `Move`. Każdy cios jest opisany poprzez pojedyncze klatki za pomocą klasy `Frame`, w których znajduje się informacja o ruchu jaki postać ma wykonać, jaki zestaw obszarów określających kolizję ma mieć wpływ na postać w danym momencie oraz jaki `sprite` ma zostać wyświetlony.

3.2.3. Wykrywanie kolizji, obsługa wejścia i wzorzec *command*

Klasy `Command`, `CommandFactory`, `DoNothingCommand`, `BlockStunCommand`, `HitStunCommand`, `SetHighStunCommand`, `MoveFighterCommand`, `StartGameCommand` wdrażają wzorzec programowania *command* (polecenia, [12]), umożliwiając abstrakcyjne reprezentacje akcji, które mogą być wykonane w grze. `Command` jest interfejsem dla wszystkich komend, `CommandFactory` jest fabryką do tworzenia instancji komend, a pozostałe klasy są konkretnymi komendami, które mogą być wywołane przez graczy lub system gry. Klasa `InputHandler` zajmuje się obsługą wejścia od użytkownika - tworzy komendy na podstawie przycisku naciśniętego przez gracza oraz aktualnego stanu postaci. `Collision` zarządza detekcją kolizji między postaciami w grze oraz wpływa na stan tych postaci. `Multiplayer` odpowiada za implementację funkcjonalności sieciowych, umożliwiając grę wieloosobową poprzez zarządzanie połączeniami i komunikacją między graczami. Gdy postać jest sterowana przez gracza połączonego przez internet to właśnie ta klasa podaje komendy do wykonania przez jego awatar wpływając na działanie `InputHandler'a`.

3.3. Fragmenty kodu

W tej sekcji zostaną omówione fragmenty kodu, które zostały uznane przez autora pracy inżynierskiej za kluczowe w działaniu programu.

3.3.1. Interfejs `Command`

Na listingu 3.1 zaprezentowano interfejs, który jest implementowany przez wiele innych klas. Interfejs ten nakazuje implementującym go klasom stworzenie metody `execute`, która najczęściej zmieniać ma stan jakiegoś obiektu (np. z klasy `Fighter`) oraz metody `undo`, mającej za zadanie wycofanie zmian wprowadzonych metodą `execute`. Dzięki temu podejściu można na przykład łatwo zmieniać przypisane do przycisków zadania lub stworzyć historię wciśniętych przycisków, a także system powtórek bazujących na logice *undo*. Dodatkowo, tworzenie obiektów reprezentujących wykonywane akcje ułatwia proces tworzenia kodu sieciowego.

Listing 3.1: Interfejs `Command`

```
public interface Command {
    void execute(Entity entity);
    void undo(Entity entity);
}
```

3.3.2. Klasa `Fighter`

Klasa `Fighter` zarządza stanem i zachowaniem postaci w grze. Zawiera informacje o zdrowiu, listy obszarów trafień (*hitboxes*) i obszarów, które mogą zostać trafione (*hurtboxes*), oraz zestaw ruchów (*movelist*). Wykorzystuje ona wzorzec programowania *update*. Sprowadza się on do implementacji metody aktualizacji stanu, która symuluje jedną klatkę zachowania obiektu. Na listingu 3.2 można zauważyć, że metoda ta tworzy rozkaz za pomocą klasy `InputHandler`,

a następnie wykonuje go. W kolejnym kroku, jeżeli gracz uczestniczy w rozgrywce online, jego komenda jest przesyłana do drugiego gracza z pomocą klasy `Multiplayer`. Na końcu postać w grze aktualizuje wyświetlany `sprite` oraz inne informacje dotyczące aktualnej klatki animacji.

Listing 3.2: Funkcja update w klasie Fighter

```
public void update() {
    Command command = inputHandler.handleInput();
    command.execute(this);
    if (multiplayer != null && (player == Player.PLAYER1 || player ==
        ↪ Player.PLAYER2)) {
        multiplayer.sendCommand(command);
    }
    updateAnimation();
}
```

3.3.3. Klasa `OfflineGameScreen`

Listing 3.3 przedstawia implementację klasy `OfflineGameScreen`. Klasa ta wykorzystuje wzorzec pętli gry (*game loop pattern*), wraz z wzorcem aktualizacji (*update pattern*), aby zarządzać cyklem życia gry w trybie *offline*. Konstruktor inicjalizuje stan gry, przydzielając graczy i tworząc kolekcję bytów (`entities`), które będą aktualizowane i renderowane w każdym cyklu gry. Inicjalizuje również system kolizji, który jest odpowiedzialny za wykrywanie i rozwiązywanie interakcji między bytami. Klasa implementuje interfejs `Screen` z frameworka `LibGDX`, co oznacza, że musi dostarczyć implementację metody `render`, która jest wywoływana na każdą klatkę gry. Ta metoda jest rdzeniem pętli gry. Przyjmuje parametr `delta`, który reprezentuje czas, jaki upłynął od ostatniej klatki, umożliwiając aktualizacje oparte na czasie rzeczywistym. Na początku każdej klatki ekran jest czyszczony.

Metoda `updateEntities` jest wywoływana w celu zaktualizowania stanu wszystkich bytów w grze. Zawiera logikę warunkową, która decyduje, czy byty powinny być aktualizowane w stanie bezczynności (`idle`), po zakończeniu rundy (`updateRoundEnd`), czy w standardowym trybie gry, gdzie każdy byt jest aktualizowany poprzez wywołanie jego metody update. Te dwie pierwsze są oddzielone od standardowego trybu ze względu na to że w trybie online będą one mogły wykonywać się po stronie klienta, niezależnie od przychodzących przez internet informacji od drugiego gracza. Po zaktualizowaniu bytów, system kolizji jest aktualizowany, aby sprawdzić i przetworzyć wszelkie kolizje, które miały miejsce pomiędzy bytami.

Na koniec metoda `renderGame` odpowiada za narysowanie aktualnego stanu gry na ekranie.

Listing 3.3: Klasa `OfflineGameScreen`

```
public class OfflineGameScreen implements Screen {
    FightingGame game;
    private Collection<Entity> entities;
    private Collision collision;
    ...// Pozostałe tekstury, dźwięki itp

    public OfflineGameScreen(FightingGame game) {
        // Przydzielanie graczy
        game.player1.setPlayer(Player.PLAYER1);
        game.player2.setPlayer(Player.PLAYER2);

        // Inicjalizacja kolekcji bytów i kolizji
        entities = new ArrayList<>();
        entities.add(game.player1);
        entities.add(game.player2);
        collision = new Collision(game.player1, game.player2);
        ...// Pobieranie załadowanych zasobów za pomocą AssetManagera
```

```
    }

    @Override
    public void render(float delta) {
        ...
        //clear
        clearScreen();
        //update
        updateEntities();
        //render
        renderGame();
        ...
    }

        private void updateEntities() {
        if (isCountdownActive || isFightMessageActive) {
            game.player1.updateIdle();
            game.player2.updateIdle();
        } else if (isWinnerMessageActive) {
            game.player1.updateRoundEnd();
            game.player2.updateRoundEnd();
        } else {
            for (Entity entity : entities) {
                entity.update();
            }
            collision.update();
        }
    }
        ...
}
```

3.3.4. Klasa `Multiplayer`

Klasa `Multiplayer` została stworzona w celu obsługi połączenia między graczami w trybie online. Wykorzystuje ona biblioteki kryo [4], kryonet [5] oraz weupnp [1], co jest widoczne w importach klasy w listingu 3.4. Konstruktor klasy nie otwiera jeszcze żadnego połączenia, a konkretne metody obsługi połączenia są wywoływane dopiero po wybraniu przez użytkownika odpowiedniego trybu gry.

Jeśli gracz wybierze opcję hostowania meczu, oznacza to uruchomienie serwera na jego komputerze za pomocą funkcji `initializeServer`. Funkcja ta rozpoczyna od znalezienia wolnego portu, którym domyślnie jest port 54555. Następnie próbuje otworzyć go za pomocą funkcji `openPortUPnP`, która wyszukuje dostępny router i sprawdza jego zewnętrzny adres IP, a następnie tworzy mapowanie portów. Po tych krokach tworzony jest serwer i przypisywane są do niego klasy do serializacji i deserializacji. Serwer nasłuchuje różne zdarzenia, takie jak połączenie i rozłączenie drugiego gracza, które zmieniają flagi klasy `Multiplayer`. Dodatkowo, jeśli do serwera przyjdzie instancja obiektu `Command` jest ona dodawana do kolejki komend. Na podstawie tej kolejki `InputHandler` definiuje zachowanie postaci gracza połączonego przez sieć. Ostatecznie serwer jest przypisywany do portu, na którym ma pracować.

W przypadku wyboru przez użytkownika opcji dołączenia do pokoju i podania kodu uzyskanego od drugiego gracza, uruchamiana jest funkcja `initializeClient`. Podobnie jak w przypadku serwera, klient również nasłuchuje przychodzące wiadomości. Jedyną różnicą jest obsługa konkretnych komend, takich jak rozpoczęcie gry czy wiadomość o powaleniu przeciwnika. Reszta komend jest obsługiwana w podobny sposób jak na serwerze.

Klasa `Multiplayer` posiada także funkcje do zamykania serwera i klienta `closeServer` i `closeClient`, które służą do zakończenie połączenia po zakończeniu rozgrywki i w przypadku serwera również zakończenie przekierowania portów na routerze.

Listing 3.4: Klasa `Multiplayer`

```
package com.mygdx.game;

import com.esotericsoftware.kryo.Kryo;
import com.esotericsoftware.kryonet.Connection;
import com.esotericsoftware.kryonet.Listener;
import com.esotericsoftware.kryonet.Server;
import com.esotericsoftware.kryonet.Client;
...
import org.bitlet.weupnp.GatewayDevice;
import org.bitlet.weupnp.GatewayDiscover;
import org.bitlet.weupnp.PortMappingEntry;


public class Multiplayer {
    private Server server;
    private Client client;
    private int serverPort;
    private int connectedPort;
    private boolean isAttemptingConnection = false;
    private boolean isConnected = false;
    private boolean startGame = false;
    private String lastErrorMessage = null;
    ...
    String ipAddress;
    LinkedBlockingDeque<Command> commandQueue;

    public Multiplayer() {
        this.commandQueue = new LinkedBlockingDeque<>();
    }

    public void initializeServer() {
        isAttemptingConnection = true;
        serverPort = findFreePort();
        if (openPortUPnP(serverPort)) {
            System.out.println("UPnP: Port został otwarty");
        } else {
            System.out.println("UPnP: Nie udało się otworzyć portu, sprawdź
                ↪ konfigurację routera lub ustawienia firewalla");
        }
        server = new Server();
        client = null;
        configureKryo(server.getKryo());

        server.addListener(new Listener() {
            @Override
            public void connected(Connection connection) {
                isConnected = true;
                lastErrorMessage = null;
            }
            @Override
            public void received(Connection connection, Object object) {
                if (object instanceof Command) {
                    System.out.println("received: " + object);
                    handleCommand((Command) object);
                }
```

```
            }
            @Override
            public void disconnected(Connection connection) {
                isConnected = false;
                lastErrorMessage = "Disconnected";
            }
        });

        ipAddress = getPublicIpAddress();
        System.out.println("IpAddress: " + ipAddress);
        try {
            server.bind(serverPort);
            server.start();
            System.out.println(" Server started on serverPort: " +
                ↪ serverPort);
        } catch (IOException e) {
            lastErrorMessage = "Connection error: " + e.getMessage();
            e.printStackTrace();
        }
    }

    public void closeServer() {
        if (server != null) {
            server.stop();
            server = null;
            removePortUPnP(serverPort);
            System.out.println("Serwer zostal zamkniety.");
        }
    }


    public void initializeClient(String ipAddress, int port) {
        isAttemptingConnection = true;
        client = new Client();
        configureKryo(client.getKryo());

        client.addListener(new Listener() {
            @Override
            public void connected(Connection connection) {
                isConnected = true;
                lastErrorMessage = null;
            }
            @Override
            public void received(Connection connection, Object object) {
                if (object instanceof SetHighStunCommand) {
                    System.out.println("received: " + object);
                    player1IsHitStunned = true;
                }
                else if (object instanceof StartGameCommand) {
                    startGame = true;
                } else if (object instanceof Command) {
                    System.out.println("received: " + object);
                    handleCommand((Command) object);
                }
            }
            @Override
            public void disconnected(Connection connection) {
                isConnected = false;
                lastErrorMessage = "disconnected";
            }
```

```
        });

        try {
            client.start();
            //System.out.println("Klient probuje sie polaczyc z " +
                ↪ ipAddress + "/" + port);
            client.connect(5000, ipAddress, port);
            this.ipAddress = ipAddress;
        } catch (IOException e) {
            lastErrorMessage = "Connection error: " + e.getMessage();
            e.printStackTrace();
        }
    }

    public void closeClient() {
        if (client != null) {
            client.stop();
            client = null;
            System.out.println("Klient zostal zamkniety.");
        }
    }

    public void stopServer() {
        if (server != null) {
            // Usuń mapowanie portów za pomocą UPnP
            removePortUPnP(serverPort);
            server.stop();
            server.close();
            System.out.println("Serwer zatrzymany");
        }
    }

    private void handleCommand(Command command) {
        commandQueue.add(command);
    }

    public void sendCommand(Command command) {
        if (server != null) {
            System.out.println("send: " + command);
            server.sendToAllTCP(command);
        } else if (client != null) {
            System.out.println("send: " + command);
            client.sendTCP(command);
        }

    }

    public boolean openPortUPnP(int port) {
        try {
            GatewayDiscover discover = new GatewayDiscover();
            discover.discover();
            GatewayDevice d = discover.getValidGateway();

            if (d == null) {
                System.out.println("Nie znaleziono bramy UPnP!");
                return false;
            }

            System.out.println("Znaleziono brame UPnP: " + d.getModelName()
                ↪ );
```

```
            // Pobierz zewnętrzny adres IP
            String externalIPAddress = d.getExternalIPAddress();
            System.out.println("Zewnetrzny adres IP to: " +
                ↪ externalIPAddress);

            // Utwórz mapowanie portów
            boolean done = d.addPortMapping(port, port, d.getLocalAddress()
                ↪ .getHostAddress(), "TCP", "KryoNet Game Server");

            if (done) {
                System.out.println("Port przekierowany: " + port);
                return true;
            } else {
                System.out.println("Nie udalo się przekierować portu: " +
                    ↪ port);
                return false;
            }
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    public boolean removePortUPnP(int port) {
        try {
            GatewayDiscover discover = new GatewayDiscover();
            discover.discover();
            GatewayDevice d = discover.getValidGateway();
            if (d != null) {
                return d.deletePortMapping(port, "TCP");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return false;
    }
        ...
}
```

Rozdział 4

Testy

Ten rozdział poświęcono na prezentację testów przeprowadzonych w ramach projektu. Testy automatyczne zostały przeprowadzone z użyciem JUnit [2] i Mockito [6].

4.1. Testy automatyczne

4.1.1. `FighterTest`

W celu zweryfikowania poprawnego działania klasy `Fighter` napisano i przeprowadzono test sprawdzający działanie metody `update` (metoda ta zajmuje się aktualizacją stanu obiektu w każdej klatce). Implementacja testu zaczyna się od zaimportowania wszystkich potrzebnych komponentów (patrz listing 4.1). W funkcji `init`, która jest udekorowana adnotacją `@BeforeAll`, zawarte są wszystkie potrzebne inicjalizacje związane z frameworkiem `LibGDX` oraz jego uruchomieniem bez środowiska graficznego. Następnie w funkcji `setUp` tworzone są wszystkie potrzebne obiekty do testów – obiekt klasy `Fighter` oraz `GameAssetManager`.

Testowana miała być poprawność zmiany stanu obiektu z klasy `Fighter`. Należało więc zasymulować wciśnięcie przycisku, a dokładniej – zwrócenie odpowiedniego polecenia z metody `handleInput` instancji obiektu "InputHandler". Takie *mockowanie* zrealizowano poprzez zwracanie komendy *MoveFighterCommand* jeżeli wywołana zostanie funkcja `handleInput`.

Test rozpoczyna się od potwierdzenia stanu dopiero utworzonej postaci. Następnie wywoływana jest funkcja `update`, która ma za zadanie zaktualizować postać na podstawie komendy. Następnie sprawdzane jest czy prawidłowo zmienione zostały konkretne pola. Dokładniej mówiąc sprawdzane jest czy postać dopiero rozpoczyna animację (czy znajduje się w klatce o zerowym indeksie), czy poruszyła się o odpowiednią odległość, czy jej stan został odpowiednio zmieniony, czy obszary kolizji zmieniły się na te zgodne z aktualną animacją oraz czy podmieniona została wyświetlana aktualnie tekstura.

Test przebiegł pomyślnie potwierdzając wszystkie sprawdzenia.

Listing 4.1: Test metody `update` z klasy `Fighter`

```
package com.mygdx.entities;
...
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

public class FighterTest extends ApplicationAdapter {
    private Fighter fighter;
    private InputHandler mockInputHandler;
    private int initialX, initialY;
```

```
@BeforeAll
public static void init() {
    HeadlessApplicationConfiguration conf = new
        ↪ HeadlessApplicationConfiguration();
    new HeadlessApplication(new FightingGame(), conf);
    Gdx.gl = mock(GL20.class);
    Gdx.gl20 = mock(GL20.class);
}
@BeforeEach
void setUp() {
    initialX = 0;
    initialY = 0;
    mockInputHandler = mock(InputHandler.class);
    GameAssetManager gameAssetManager = new GameAssetManager();
    gameAssetManager.loadAssets();
    while (!gameAssetManager.manager.update()) {
        // oczekiwanie aż wszystkie zasoby sie załadują
    }
    fighter = new Fighter(initialX, initialY, Player.PLAYER1, 0, 0, 0,
        ↪ 0, 10, gameAssetManager.manager);
    fighter.setInputHandler(mockInputHandler);
}

@Test
void testUpdateWithMoveForwardCommand() {
    // Sprawdzenie początkowego stanu postaci
    assertEquals(0, fighter.getCurrentFrame());
    assertEquals(initialX, fighter.getX());
    assertEquals(initialY, fighter.getY());
    assertEquals(State.NEUTRAL, fighter.getState());
    Move idleMove = fighter.getMovelist().getMove(State.NEUTRAL.getId()
        ↪ );
    List<Rectangle> initialHurtboxes = idleMove.getFrame(0).
        ↪ getHurtboxes();
    List<Rectangle> initialHitboxes = idleMove.getFrame(0).getHitboxes
        ↪ ();
    TextureRegion initialSprite = idleMove.getFrame(0).getSprite();
    assertEquals(initialHurtboxes, fighter.getHurtboxes());
    assertEquals(initialHitboxes, fighter.getHitboxes());
    assertEquals(initialSprite, fighter.getTextureRegion());

    // Symulacja otrzymania komendy MoveFighterCommand od InputHandler'
        ↪ a
    when(mockInputHandler.handleInput()).thenReturn(CommandFactory.
        ↪ moveFighterCommandForward(fighter));

    // Aktualizacja logiki postaci
    fighter.update();

    // Sprawdzenie, czy aktualna klatka animacji to 0
    assertEquals(0, fighter.getCurrentFrame());

    // Sprawdzenie, czy postać poruszyła się o oczekiwaną odległość
    Move goingForwardMove = fighter.getMovelist().getMove(State.
        ↪ GOING_FORWARD.getId());
    int xAxisMovement = goingForwardMove.getFrame(0).getXAxisMovement()
        ↪ ;
    int yAxisMovement = goingForwardMove.getFrame(0).getYAxisMovement()
        ↪ ;
```

```
        assertEquals(initialX + xAxisMovement, fighter.getX());
        assertEquals(initialY + yAxisMovement, fighter.getY());

        // Sprawdzenie, czy stan postaci zmienił się na GOING_FORWARD
        assertEquals(State.GOING_FORWARD, fighter.getState());

        // Sprawdzenie, czy hurtboxy i hitboxy są zgodne z aktualną klatką
            ↪ ruchu
        List<Rectangle> hurtboxes = goingForwardMove.getFrame(0).
            ↪ getHurtboxes();
        for (Rectangle hurtbox : hurtboxes) {
            // Jako że położenie hurtboxów jest względne (wobec postaci)
            // to należy jeszcze dodać przebytą drogę
            hurtbox.x = hurtbox.x + xAxisMovement;
        }
        List<Rectangle> hitboxes = goingForwardMove.getFrame(0).getHitboxes
            ↪ ();
        for (Rectangle hitbox : hitboxes) {
            // Jako że położenie hitboxów jest względne (wobec postaci)
            // to należy jeszcze dodać przebytą drogę
            hitbox.x = hitbox.x + xAxisMovement;
        }
        assertEquals(hurtboxes ,fighter.getHurtboxes());
        assertEquals(hitboxes ,fighter.getHitboxes());

        // Sprawdzenie czy postac zmienilła wyświetlaną teksturę
        TextureRegion sprite = goingForwardMove.getFrame(0).getSprite();
        assertEquals(sprite, fighter.getTextureRegion());
    }
}
```

4.1.2. CollisionTest

Test polega na symulowaniu serii klatek gry, w których atakujący wykonuje atak wysoki (high attack), a odbierający porusza się do przodu. Celem jest sprawdzenie, czy w odpowiedniej klatce dochodzi do kolizji i czy system poprawnie reaguje na tę sytuację, zmieniając stan postaci odbierającej na `HIT_STUNNED_HIGH` oraz czy poprawnie odejmuje jej zdrowie.

Listing 4.2: Test poprawnego wykrywania kolizji w klasie `Collision`

```
package com.mygdx.engine;
...
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class CollisionTest extends ApplicationAdapter {
    Collision collision;
    private Fighter attacker, receiver;
    private InputHandler attackerMockInputHandler, receiverMockInputHandler
        ↪ ;
    private int attackerInitialX, attackerInitialY, receiverInitialX,
        ↪ receiverInitialY;
    @BeforeAll
    public static void init() {
        HeadlessApplicationConfiguration conf = new
            ↪ HeadlessApplicationConfiguration();
        new HeadlessApplication(new FightingGame(), conf);
        Gdx.gl = mock(GL20.class);
        Gdx.gl20 = mock(GL20.class);
```

```
}
@BeforeEach
void setUp() {
    // Początkowe rozmieszczenie umieszcza postacie tuż obok siebie
    attackerInitialX = 0;
    attackerInitialY = 0;
    receiverInitialX = 300;
    receiverInitialY = 0;
    GameAssetManager gameAssetManager = new GameAssetManager();
    gameAssetManager.loadAssets();
    while (!gameAssetManager.manager.update()) {
        // oczekiwanie aż wszystkie zasoby sie zaladują
    }
    attacker = new Fighter(attackerInitialX, attackerInitialY, Player.
        ↪ PLAYER1, 0, 0, 0, 0, 10, gameAssetManager.manager);
    attackerMockInputHandler = spy(new InputHandler(attacker, Player.
        ↪ PLAYER1, 0, 0, 0, 0, 10));
    attacker.setInputHandler(attackerMockInputHandler);
    receiver = new Fighter(receiverInitialX, receiverInitialY, Player.
        ↪ PLAYER2, 0, 0, 0, 0, 10, gameAssetManager.manager);
    receiverMockInputHandler = spy(new InputHandler(receiver, Player.
        ↪ PLAYER2, 0, 0, 0, 0, 10));
    receiver.setInputHandler(receiverMockInputHandler);
    collision = new Collision(attacker, receiver);
}

@Test
void testCollisionDetectionAndAftermaths() {
    // Symulacja otrzymania komendy AttackCommand od InputHandler'a
    doAnswer(invocation -> {
        Command command = CommandFactory.AttackCommandHigh(attacker);
        attackerMockInputHandler.getCommandHistory().add(command);
        attacker.setCurrentFrame(attacker.getCurrentFrame() + 1);
        return command;
    }).when(attackerMockInputHandler).handleInput();
    // Symulacja funkcjonalności InputHandler.handleInput() ograniczają
        ↪ cej się
    // do zwracania komendy MoveFighterCommand lub HitStunCommand w
        ↪ przypadku,
    // gdy Collision wykryje otrzymanie ataku high
    doAnswer(invocation -> {
        Command command = CommandFactory.moveFighterCommandForward(
            ↪ receiver);
        int currentFrame = receiver.getCurrentFrame() + 1;
        if (receiver.isHitStunnedHigh()) {
            if (receiver.getState() != State.HIT_STUNNED_HIGH) {
                receiver.setCurrentFrame(0);
                command = CommandFactory.hitStunCommandHigh(receiver);
                receiverMockInputHandler.getCommandHistory().add(
                    ↪ command);
                return command;
            } else if (receiver.getState() == State.HIT_STUNNED_HIGH &&
                ↪  currentFrame >= receiver.getMovelist().getMove(State
                ↪ .HIT_STUNNED_HIGH.ordinal()).getFrameCount()) {
                receiver.setHitStunnedHigh(false);
            } else {
                receiver.setCurrentFrame(currentFrame);
                command = CommandFactory.hitStunCommandHigh(receiver);
                receiverMockInputHandler.getCommandHistory().add(
                    ↪ command);
```

```
                    return command;
                }
            }
            receiverMockInputHandler.getCommandHistory().add(command);
            receiver.setCurrentFrame(currentFrame);
            return command;
        }).when(receiverMockInputHandler).handleInput();

        // Symulacja upływanego czasu, gdzie "i" oznacza indeks aktualnej
            ↪ klatki
        for (int i = 0; i <= 14; i++) {
            attacker.update();
            receiver.update();
            collision.update();
            if (i <= 12) {
                // Sprawdzanie czy receiver nie dostał za wcześnie atakiem
                assertFalse(receiver.isHitStunnedHigh());
            } else if (i == 13) {
                // W 14 klatce collision powinien wykryć kolizję i zmienić
                    ↪ flagę isHitStunnedHigh na true
                assertTrue(receiver.isHitStunnedHigh());
            } else if (i >= 14) {
                // W 15 klatce receiver powinien odczuć skutki dostania
                    ↪ atakiem high
                int damage = attacker.getMovelist().getMove(State.
                    ↪ HIGH_ATTACK.getId()).getDamage();
                assertEquals(receiver.getMaxHealth() - damage, receiver.
                    ↪ getHealth());
                assertEquals(State.HIT_STUNNED_HIGH, receiver.getState());
            }
        }
    }
}
```

4.2. Testy manualne

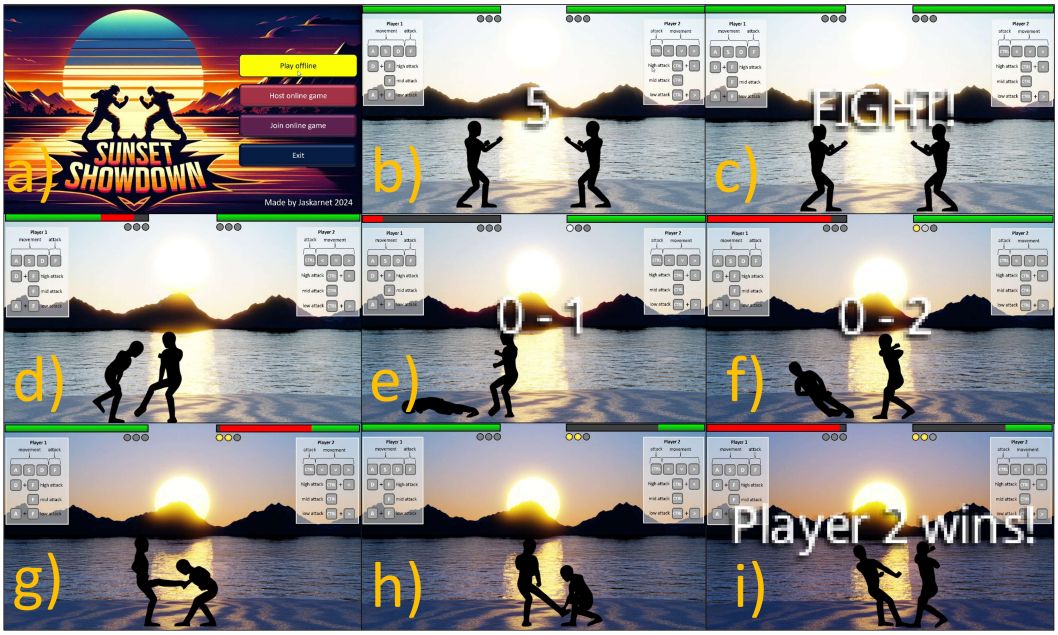
Testy te można zaliczyć do grupy testów akceptacyjnych, choć bez formalnie wyspecyfikowanego scenariusza testów. Generalnie testy te miały dowieść, czy aplikacja spełnia oczekiwania użytkownika, a więc czy zaimplementowana gra pozwala na „bijatykę”.

4.2.1. Rozgrywka w trybie offline

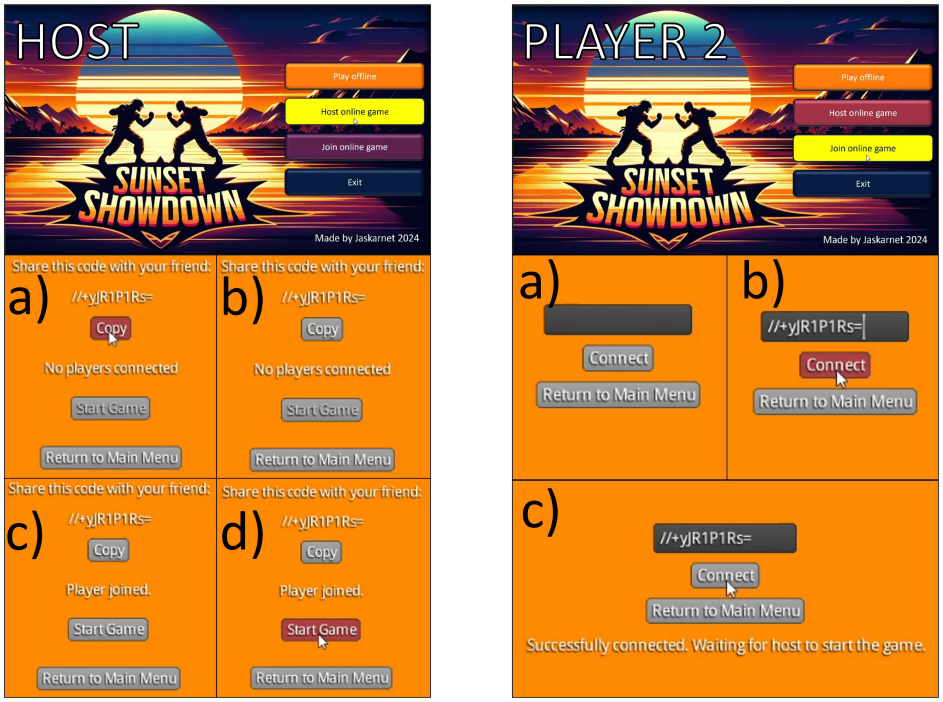
Test polegał na uruchomieniu programu oraz wybraniu opcji "Play offline"z menu głównego. Następnie sprawdzone zostały wszystkie dostępne ruchy postaci oraz ich interakcje z przeciwnikiem. Wraz z postępem rozgrywki przetestowany został również system rund. Test wykonano wielokrotnie w celu sprawdzenia, czy dla każdego przypadku wypisywane są odpowiednie wiadomości wskazujące aktualny wynik lub wygranego całego pojedynku (lub remisu). Zrzuty ekranu z przykładowej rozgrywki pokazano na rysunku 4.1. Podczas całego testu wszystkie zachowania postaci oraz wiadomości gry zgadzały się z oczekiwaniami.

4.2.2. Rozgrywka w trybie online

W celu sprawdzenia poprawności działania trybu online przeprowadzono testy polegające na rozegraniu gry z dwóch różnych komputerów korzystając z opcji dostępnych w programie. Proces wyglądał następująco: host zakładał pokój, a następnie udostępniał drugiemu graczowi kod



Rys. 4.1: Przetestowanie trybu offline (chronologia zgodna z kolejnością alfabetyczną)



Rys. 4.2: Kolejne etapy tworzenia pokoju przez hosta oraz dołączania do niego przez drugiego gracza

zaproszeniowy. Drugi gracz po wpisaniu kodu w odpowiednim miejscu łączył się z przeciwnikiem i jeżeli połączenie było udane obydwaj dostawali odpowiedni komunikat. Host po kliknięciu przycisku startował grę, po której zaczynała się rozgrywka. Na rysunku 4.2 pokazano zrzuty ekranu odpowiadające etapom łączenia się, zarówno z punktu widzenia hosta jak i gracza dołączającego. Sama gra prezentowała się podobnie do rozgrywki offline z paroma drobnymi różnicami dotyczącymi wyświetlanych komunikatów. Postacie odpowiednio reagowały na sterowanie graczy. Przebieg rund jak i całego meczu odbywał się prawidłowo. Całość testu potwierdza działanie kodu sieciowego.

Rozdział 5

Podsumowanie

Zgodnie z założeniami pracy dyplomowej, stworzono projekt gry z gatunku bijatyk 2D umożliwiającej prowadzenie rozgrywek online.

Niniejsza praca dyplomowa skupiała się na procesie projektowania, implementacji i testowania dwuwymiarowej gry bijatyki z funkcjonalnością rozgrywki online. W ramach pracy, autor dokonał przeglądu istotnych koncepcji i technik wykorzystywanych w projektowaniu gier, szczególnie tych z gatunku bijatyk. Kładąc szczególny nacisk na aspekty techniczne, takie jak system kolizji, animacja postaci i zarządzanie rozgrywką multiplayer, autor przedstawia proces od koncepcji po realizację. W części praktycznej, opisano rozwiązania wykorzystane do napisania kodu gry. Praca obejmuje również podejście do testowania, co jest kluczowe dla zapewnienia jakości i płynności rozgrywki. Projekt ten okazał się dla autora nie tylko możliwością zastosowania teoretycznej wiedzy w praktyce, ale również platformą do rozwijania umiejętności programistycznych i zdobywania doświadczenia w tworzeniu gier. Praca ta podkreśla, jak ważne w procesie edukacyjnym jest praktyczne doświadczenie, które pozwala na głębsze zrozumienie i opanowanie skomplikowanych aspektów tworzenia gier.

W trakcie procesu testowania programu na różnych konfiguracjach komputerowych, autor projektu napotkał problem związany ze złym wyświetlaniem tekstur. Symptomem tego problemu było pojawianie się czarnego prostokąta w miejscu niektórych animacji. Po dokładnej analizie potencjalnych przyczyn, stwierdzono, że kwestia ta dotyczy głównie dłuższych animacji. Ustalono, że długość animacji ma bezpośredni wpływ na rozmiar *spritesheet'u* zapisywanego w zasobach gry. Dotychczasowe założenie, że każdy wiersz *spritesheet'u* zawiera jedynie pojedynczy *sprite*, okazało się niewystarczające przy animacjach wymagających znacznej liczby *sprite'ów*, co prowadziło do tworzenia nieproporcjonalnie długich obrazków. Problem ten okazał się szczególnie zauważalny w przypadku architektury niektórych komputerów oraz wersji OpenGL zainstalowanej na tych systemach. W odpowiedzi na wykryte problemy, autor zdecydował się na modyfikację wszystkich *spritesheet'ów*, ustalając, że każdy wiersz będzie zawierał 8 *sprite'ów*, przy czym każdy *spritesheet* składa się z 8 takich wierszy. Taki kwadratowy układ *spritesheet'ów* skutecznie rozwiązał problem z wyświetlaniem tekstur na dotychczas problematycznych konfiguracjach sprzętowych.

Początkowy zamiar autora zakładał implementację *Rollback Netcode* [11] w trybie multiplayer, jednak ze względu na ograniczenia czasowe, ten element nie został w pełni zrealizowany. *Rollback Netcode* miał na celu symulowanie ruchów gracza w sytuacjach braku aktualnych danych o jego działaniach i korygowanie stanu gry po otrzymaniu tych informacji. W obecnej wersji kodu widoczne są początki implementacji tej funkcjonalności, gdzie wszystkie komendy skierowane do postaci gracza są rejestrowane w strukturze danych typu *circular buffer*, z możliwością ich cofnięcia. W przyszłości można rozważyć rozbudowę tej funkcjonalności o bardziej zaawansowany model przewidywania ruchów gracza.

Literatura

[1] A. Bahgat. weupnp Library GitHub Repository. https://github.com/bitletorg/weupnp [dostęp dnia 24 stycznia 2024].

[2] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, J. de Rancourt, C. Stein. JUnit 5 User Guide - Version 5.10.1. https://junit.org/junit5/docs/current/user-guide/ [dostęp dnia 24 stycznia 2024].

[3] Blender Foundation. Blender 4.0 Reference Manual. https://docs.blender.org/manual/en/latest/ [dostęp dnia 24 stycznia 2024].

[4] Esoteric Software. Kryo Serialization Framework GitHub Repository. https://github.com/EsotericSoftware/kryo [dostęp dnia 24 stycznia 2024].

[5] Esoteric Software. Kryonet Networking Library GitHub Repository. https://github.com/EsotericSoftware/kryonet [dostęp dnia 24 stycznia 2024].

[6] S. Faber, B. Dutheil, R. Winterhalter, T. van der Lippe, M. Grzejszczak. Mockito Testing Framework - Official Site. https://site.mockito.org/ [dostęp dnia 24 stycznia 2024].

[7] Git community. Git Documentation. https://git-scm.com/doc [dostęp dnia 24 stycznia 2024].

[8] GitHub, Inc. GitHub Docs. https://docs.github.com/en [dostęp dnia 24 stycznia 2024].

[9] ImageMagick Studio LLC. Annotated List of Command-line Options. https://imagemagick.org/script/command-line-options.php [dostęp dnia 24 stycznia 2024].

[10] JetBrains s.r.o. Getting Started with IntelliJ IDEA - Official Guide. https://www.jetbrains.com/help/idea/getting-started.html [dostęp dnia 24 stycznia 2024].

[11] A. Lioret, L. Diler, S. Dalil, M. Mota. Hybrid Prediction for Games' Rollback Netcode, july 2022. https://www.academia.edu/89865288/Hybrid_Prediction_for_Games_Rollback_Netcode [dostęp dnia 23 stycznia 2024].

[12] R. Nystrom. *Game programming Patterns*. Robert Nystrom, 2014.

[13] OpenJFX Community. JavaFX Documentation. https://openjfx.io/ [dostęp dnia 24 stycznia 2024].

[14] Pixabay. Niesamowite darmowe obrazy. https://pixabay.com/pl/ [dostęp dnia 24 stycznia 2024].

[15] H. Schildt. *Java. Kompendium programisty*. Helion, wydanie 2, 2023.

[16] M. Zechner. LibGDX Wiki - The Official Documentation. https://libgdx.com/wiki/ [dostęp dnia 24 stycznia 2024].

Dodatek A

Instrukcja wdrożeniowa

A.1. Uruchomienie z pliku wykonywalnego

Aby pobrać plik wykonywalny gry należy wejść na stronę repozytorium `https://github.com/Jaskarnet/2DFightingGame.git`, następnie przejść do sekcji `release`, gdzie można znaleźć plik `.rar`. Po jego pobraniu i rozpakowaniu wystarczy uruchomić znajdujący się w nim plik wykonywalny `SunsetShowdown.exe`.

A.2. Skompilowanie kodu

Wymagania wstępne: Java Development Kit (JDK), Gradle, środowisko programistyczne – zalecane jest użycie IntelliJ IDEA, git – potrzebny do klonowania repozytorium.
Klonowanie repozytorium: Otwórz terminal. Sklonuj repozytorium, używając polecenia: `git clone https://github.com/Jaskarnet/2DFightingGame.git`. Przejdź do sklonowanego katalogu.
Konfiguracja projektu: Otwórz sklonowany projekt w środowisku programistycznym. Upewnij się, że struktura projektu została poprawnie zaimportowana i że wszystkie potrzebne zależności są zainstalowane. Możliwe, że środowisko samo wykryje i zainstaluje potrzebne zależności.
Uruchamianie gry: Znajdź główną klasę aplikacji, która znajduje się w ścieżce `desktop/src/com/mygdx/game/DesktopLauncher.java`. Uruchom projekt, korzystając z przycisku *Run* w IDE. Jeżeli uruchomienie programu zakończy się błędem, prawdopodobnie należy ustawić katalog roboczy na folder `assets`. W tym celu należy nacisnąć przycisk *More Actions* (trzy kropki) znajdujący się w górnej części graficznego interfejsu obok nazwy uruchamianej konfiguracji i z menu kontekstowego wybrać opcję *Edit* z zakładki *Configuration*. Następnie w otwartym okienku należy znaleźć opcję *Working Directory* i zmienić jej wartość na ścieżkę do folderu `assets` znajdującego się pod głównym folderem projektu. Po tej konfiguracji program powinien się skompilować i uruchomić poprawnie.
Uwagi końcowe: Dokładne kroki mogą się różnić w zależności od konkretnego środowiska programistycznego i konfiguracji systemu. Jeśli środowisko automatycznie nie pobrało zależności, należy je uwzględnić.

Dodatek B

Opis załączonej płyty CD/DVD

Na płycie CD-R zapisano dwa foldery o nazwie *Opis gry* oraz *Gra*. W folderze *Opis gry* umieszczono dokument `pdf` z opisem pracy inżynierskiej pod tytułem *Projekt gry z gatunku bijatyk 2D umożliwiającej prowadzenie rozgrywek online*. W drugim folderze o nazwie *Gra* umieszczono kod źródłowy gry *Sunset Showdown*.