



POLITECHNIKA  
LUBELSKA  
WYDZIAŁ ELEKTROTECHNIKI  
I INFORMATYKI

# Praca dyplomowa Inżynierska

Na kierunku: Informatyka

W specjalności: Inżynieria Oprogramowania

**Projekt i implementacja gry platformowej w środowisku Unity**

**Design and implementation of a platform game in the Unity Environment**

Marcin Kamiński

Nr. Albumu: 95430

Grzegorz Kondratowicz-Kucewicz

Nr. Albumu: 95443

Jakub Król

Nr. Albumu: 95456

Promotor: dr inż. Jakub Smołka



## **Streszczenie**

Praca inżynierska ma na celu implementację gry platformowej w środowisku Unity w oparciu o utworzoną wcześniej dokumentację projektową. Jest to gra dwuwymiarowa o tematyce warcabów oraz szachów. Początkowym etapem pracy było obmyślenie fabuły oraz funkcjonalności takich jak opcje dostępne w grze oraz jej przebieg. Dzięki wcześniej ustalonej oprawie graficznej i opisie postaci, gra ma zachować spójność i zapewnić przyjemną rozgrywkę. Modele wykorzystane w grze zostały utworzone w programie Blender, dzięki czemu zachowują one naturalną głębię. Tekstury wykorzystane w grze pochodzą z ogólnodostępnych źródeł, a do utworzenia złożonych animacji wykorzystano narzędzie Mixamo, które pozwala automatycznie wygenerować szkielet dla modelu postaci. Skrypty gry zostały napisane w języku C# za pomocą środowiska Visual Studio. Aby zachować kontrolę nad postępem prac i umożliwić współpracę zespołu, wykorzystano repozytorium Plastic SCM do zarządzania wersjami projektu. Projekt łączy elementy popularnych gier planszowych z dynamicznym środowiskiem gier platformowych, zapewniając graczom nowe doświadczenia.

## **Abstract**

The Engineering Thesis aims to implement a 2D platform game in the Unity environment based on previously created project documentation. The game is themed around checkers and chess. The initial phase of the project involved devising the storyline and functionalities, including in-game options and progression. Leveraging pre-established graphic design and character descriptions, the game seeks to maintain coherence and offer an enjoyable gaming experience. Models used in the game were crafted in Blender, preserving a natural depth. Textures come from readily available sources, and complex animations were created using Mixamo, which allows for the automatic generation of character skeletons. Game scripts were written in C# within the Visual Studio environment. To monitor progress and facilitate team collaboration, the Plastic SCM repository was utilized for version control. This project combines elements of popular board games with the dynamic environment of platform games, providing players with a fresh gaming experience.



# **Spis treści**

1. Wstęp	1
2. Cel, zakres i podział pracy	2
2.1. Cel pracy	2
2.2. Zakres pracy	2
2.3. Podział pracy	2
3. Analiza rynku	4
3.1. Ogólna analiza rynku	4
3.2. Gry o tematyce szachów i warcabów	4
3.2.1. Shotgun King: The Final Checkmate	4
3.2.2. Pawnbarian	5
3.3. Gry Platformowe	5
3.3.1. New Super Mario Bros	5
3.3.2. Cuphead	6
4. Opis wymagań systemu	8
4.1. Wymagania funkcjonalne	8
4.2. Wymagania niefunkcjonalne	9
4.3 Definiowanie wymagań niefunkcjonalnych	9
4.3.1. Wymagania systemowe:	9
4.3.2. Wymagania dot. Aplikacji	9
4.4 Tworzenie drzewa funkcji	10
5. Modelowanie wymagań funkcjonalnych	11
5.1. Tworzenie diagramów BPMN	11
5.2 Tworzenie diagramu DFD	15
5.3 Definiowanie wymagań funkcjonalnych	16
6. Przechowywanie danych gry	19
6.1. Pliki JSON	19
6.2. Klasa "PlayerPrefs"	19
6.3. Zasoby aplikacji	19
7. Opis elementów gry	21
7.1. Opis głównej postaci	21
7.2. Opis przeciwników podstawowych	21
7.2.1. Biały Pion	21

7.2.2. Biały Skoczek	21
7.2.3. Biały Goniec	22
7.2.4. Biała Wieża	22
7.2.5. Czarny Pion	22
7.2.6. Czarny Skoczek	23
7.2.7. Czarna Wieża	23
7.3. Opis przeciwników specjalnych	24
7.3.1. Leniwy Naczelnik Więzienia	24
7.3.2. Rozjuszony Naczelnik Więzienia	24
7.3.3. Biały Król	25
7.3.4. Czarny Król	25
7.4. Opis Pułapek	26
7.4.1. Kolce	26
7.4.2. Piła tartaku	26
8. Zarys fabuły	27
8.1. Krótki opis fabularny	27
8.2. Zarys świata	27
8.3. Prolog	27
8.4. Akt I - Ucieczka	27
8.5. Akt II - Pościg	27
8.6. Akt III - Bezpieczna przystań	27
8.7. Akt IV - Zemsta	28
8.8. Akt V - Gorzka prawda	28
8.9. Epilog	28
9. Projektowanie widoków aplikacji	29
10. Projekt i implementacja poziomów	33
10.1. Tworzenie poziomów w Unity	33
10.2. Poziom 1 - Więzienie	33
10.3. Poziom 1 - Walka z Naczelnikiem Więzienia	34
10.4. Poziom 2 - Ucieczka przez wsie	35
10.5. Poziom 2 - Ponowna walka z Naczelnikiem	35
10.6. Poziom 3 - Wędrownka przez las	35
10.7. Poziom 4 - Podróż przez miasto	36
10.8. Poziom 4 - Walka w katedrze z białym królem	36

10.9. Poziom 5 - Walka w obozie - Walka z czarnym królem	37
11. Zastosowane technologie	38
11.1. Środowisko Unity	38
11.2. Język C# i środowisko Microsoft Visual Studio	38
11.3. Program graficzny Blender	39
11.4. Platforma Mixamo	39
11.5. Środowisko Visual Paradigm	39
11.6. System kontroli wersji Plastic SCM	39
12. Organizacja pracy w środowisku UNITY	40
13. Implementacja	44
13.1. Stworzenie projektu w systemie kontroli wersji	44
13.2. Tworzenie modeli i animacji	44
13.3 Pisanie skryptów	45
13.4 Przechowywanie danych gry	45
13.5 Postać główna	47
13.6 Menu i ustawienia	54
13.7 System punktacji	63
14 Ogólne ustawienia przeciwników	67
14.1. Otrzymywanie obrażeń przez przeciwników	67
14.2. Niszczenie obiektu przeciwnika	68
15 Implementacja przeciwników	70
15.1. Implementacja przeciwników podstawowych	70
15.1.1. Biały Pion	70
15.1.2. Biały Skoczek	76
15.1.3. Biały Goniec	79
15.1.4. Biała Wieża	81
15.1.5. Czarny Pion	84
15.1.6. Czarny Skoczek	84
15.1.7. Czarny Goniec:	85
15.1.8. Czarna Wieża	89
15.3 Implementacja przeciwników specjalnych	91
15.3.1. Leniwy Naczelnik Więzienia	91
15.3.2. Rozjuszony Naczelnik Więzienia	106
15.3.3. Biały Król	110

15.3.4. Czarny Król	114
15.4. Pułapki	116
16. Implementacja Widoków Aplikacji	117
16.1. Scena Menu głównego	117
16.2. Ekran wyboru profilu gracza	117
16.3. Scena wyboru poziomu	118
16.4. Scena Ustawień	119
16.5. Scena przykładowego poziomu	120
17 Testy	123
18 Wnioski	124
Bibliografia	125

## 1. Wstęp

Według powszechniej opinii Polaków warcaby są uznawane za gorszą, mniej skomplikowaną wersję szachów. Jednak mimo to popularność warcabów i szachów na przestrzeni lat była zbliżona.

W ostatnich latach poprzez wybuch pandemii czy też powstanie serialu "Gambit Królowej" znacząco wzrosła popularność szachów co przełożyło szalę na korzyść tej gry logicznej. Na podstawie artykułu zawartego na stronie "chess.com" [1] w 2017 roku platforma miała ok 20 mln zarejestrowanych użytkowników, gdzie w grudniu 2022 roku strona "chess.com" poinformowała o przekroczeniu 100 mln zarejestrowanych użytkowników. Oznacza to ponad pięciokrotny wzrost liczby użytkowników w okresie pięciu lat. Niestety nie przełożyło się to na popularność warcabów i jest to powodem, dla którego został utworzony projekt. Biorąc pod uwagę niewielki odsetek gier tego typu, które w sposób atrakcyjny przedstawiają klasyczne, logiczne rozgrywki, projekt będzie unikalny na rynku, wzbudzając tym samym zainteresowanie u potencjalnych użytkowników. Jako grupę docelową została obrana młodzież, która zna szachy jednak możliwe że nie została zaznajomiona z warcabami.

Gra stworzona w ramach projektu należy do gatunku gier platformowych, w których gracz porusza się po dwuwymiarowych etapach. W tego typu produkcjach każdy etap złożony jest zazwyczaj ze zróżnicowanego terenu oraz przeciwników. Zadaniem gracza jest pokonanie przeszkód oraz dotarcie do punktu kontrolnego. Pokonywanie kolejnych poziomów będzie skutkowało w progresji w fabule.

## **2. Cel, zakres i podział pracy**

### **2.1. Cel pracy**

Celem pracy inżynierskiej jest zaprojektowanie oraz zaimplementowanie gry komputerowej z gatunku gier platformowych w środowisku Unity. Utworzenie gry o tematyce warcabów oraz szachów ma na celu spopularyzować te od wielu pokoleń znane gry wśród dzieci i młodzieży.

### **2.2. Zakres pracy**

Zakres pracy wymaganej do realizacji projektu:

- Analiza rynku
- Zdefiniowanie wymagań funkcjonalnych i niefunkcjonalnych
- Utworzenie diagramów BPMN (przedstawienie przepływu czynności) oraz DFD (model przepływu informacji)
- Opracowanie projektu gry - planu poziomów, fabuły, przeciwników i funkcji gry
- Stworzenie projektu gry w środowisku Unity wraz z wyborem odpowiedniego systemu kontroli wersji
- Wybranie gotowych oraz stworzenie własnych zasobów dla projektu takich jak: modele, tekstury, tła, animacje i dźwięki
- Przygotowanie skryptów w języku C# oraz naniesienie ich na odpowiednie obiekty
- Implementacja gry w oparciu o stworzony wcześniej zestaw assetów (Budowa poszczególnych scen tj.: menu główne, ustawienia, poziomy)
- Testowanie gry pod kątem ewentualnych błędów oraz ich korekta
- Tworzenie dokumentacji projektowej

### **2.3. Podział pracy**

Grupa projektowa składała się z 3 osób realizujących projekt według ustalonego wcześniej harmonogramu podziału prac, co w pozytywny sposób odbiło się na realizacji pracy. Niektóre zadania były realizowane wspólnie przez różne osoby, inne natomiast były pracą indywidualną poszczególnych członków zespołu projektowego. W poniżej tabeli (Tab.2.1.) został przedstawiony podział prac zespołu.

Tabela 2.1. - Harmonogram prac realizacji projektu

Lp.	Zadanie	Osoba realizująca
1	Definiowanie wymagań funkcjonalnych oraz niefunkcjonalnych	MK, JK
2	Modelowanie wymagań funkcjonalnych oraz niefunkcjonalnych	MK (diagram DFD, drzewo funkcji) JK (Modelowanie wymagań, diagramy BPMN)
3	Zarys fabularny	JK i MK (zarys fabularny)
4	Opis Wybranych Technologii	GKK
5	Opis elementów gry	JK, MK
6	Analiza Rynku	GKK
7	Projektowanie widoków aplikacji	MK
8	Budowanie scen w Unity	JK, MK, GKK
9	Projekt poziomów	GKK
10	Sformatowanie dokumentacji projektu	GKK
11	Tworzenie modeli	MK, JK
12	Tworzenie Animacji	MK, JK
13	Renderowanie grafik	GKK
14	Teksturowanie	GKK, JK, MK
15	Tworzenie skryptów C#	JK, MK
16	Wnioski	MK, JK

**Legenda:**

- MK - Marcin Kamiński
- GKK - Grzegorz Kondratowicz-Kucewicz
- JK - Jakub Król

## 3. Analiza rynku

### 3.1. Ogólna analiza rynku

Gatunek gier platformowych jest z jednym z najstarszych i najliczniejszych gatunków wśród gier komputerowych. Jego historia zaczyna się wraz z 1980r. i wydaną wtedy grą automatową „Space Panic”, natomiast wydany w 1981r. „Donkey Kong” przypieczętował założenia gatunku. Proste zasady oraz nieskomplikowane mechaniki uczyniły gry platformowe łatwo dostępymi i przyjemnymi dla graczy o różnym poziomie zaawansowania. Gatunek również i dziś cieszy się nie małą popularnością, najlepiej pokazuje to fakt że w 2022r. na samej platformie Steam zostało wydanych 1525 gier skategoryzowanych jako gry platformowe 2D oraz 861 gier opisanych jako gry platformowe 3D. Dzisiaj gracze mogą wybierać spośród setek gier dostępnych na rynku o różnych motywach i tematach, jednak na rynku aktualnie nie ma żadnej gry platformowej, która tematem nawiązywałaby do szachów i warcabów.

### 3.2. Gry o tematyce szachów i warcabów

Zarówno szachy jak i warcaby to gry znane już od setek lat. Ich popularność na całym świecie sprawiła że niemal każda osoba zna te gry i rozumie przynajmniej postawowe zasady. Z faktu tego korzystają twórcy gier, którzy wykorzystując szachy i warcaby jako podstawę dla swoich produkcji. Tworzą oni unikalne i ciekawe gry, które stawiają te wiekowe gry planszowe w nowym świetle. Przykładami takich gier mogą być:

#### 3.2.1. *Shotgun King: The Final Checkmate*

„Shotgun King” to wydana w maju 2022r. gra typu „rougelike” z elementami gry logicznej. Gracz wciela się w czarnego króla wyposażonego w strzelbę, a celem każdego poziomu jest zbitie białego króla. Gra posiada mechaniki przejmowania przez gracza ruchów innych, uprzednio zbitych figur szachowych, a tłem dla rozgrywki jest prosta liniowa fabuła. Gra zebrała bardzo dobre recenzje zarówno od krytyków jak i graczy, co najlepiej obrazuje fakt że na platformie Steam 93% wszystkich recenzji jest pozytywna. Wygląd gry został zaprezentowany na rysunku 3.1.



Rys. 3.1. - Zrzut ekranu z gry „Shotgun King”  
Zródło: [https://store.steampowered.com/app/1972440/  
Shotgun\\_King\\_The\\_Final\\_Checkmate/](https://store.steampowered.com/app/1972440/Shotgun_King_The_Final_Checkmate/)

### 3.2.2. *Pawnbarian*

„Pawnbarian” to prosta gra typu „rouglike” wydana w 2021 roku. W „Pawnbarian” gracz kieruje pionem, którego ruchy ustala za pomocą kart ruchu które odpowiadają poszczególnym figurom szachowym. Celem gry jest oczyszczenie niewielkiej planszy - Lochu z innych wrogich pionów. Gra posiada prosty, acz wciągający schemat oraz jest przystępna dla większości graczy, między innymi dzięki prostej grafice i interfejsowi (rys 3.2.). Zwiększający się z czasem rozgrywki poziom trudności stanowi dla gracza dodatkowe wyzwanie. Gra zdobyła pozytywne recenzje, z czego 92% użytkowników na platformie Steam wyraziło pozytywne opinie.



Rys. 3.2. - Zrzut ekranu z gry „Pawnbarian”  
Zródło: <https://store.steampowered.com/app/1142080/Pawnbarian/>

## 3.3. Gry Platformowe

Gry platformowe stanowią znaczący udział w rynku gier [2]. Jednymi z koronnych przykładów gier platformowych mogą być:

### 3.3.1. *New Super Mario Bros*

„New Super Mario Bros” to wydana przez Nintendo w 2006 roku gra platformowa na konsolę Nintendo DS. Gra należy do kultowej już serii „Super Mario Bros” i w projekcie poziomów nawiązuje do oryginalnego „Super Mario Bros” z 1985 roku. Gra posiada linową fabułę, która jest podzielona na 8 aktów - przypisanych poszczególnym światom, z których każdy posiada ok. 10 poziomów. Gra posiada proste reguły i mechaniki oraz niski poziom trudności. Wyjątkowym dla gier napisanych dla platformy Nintendo DS jest obecność w konsoli dwóch ekranów: górnego i dolnego, co pozwala twórcom na zaprojektowanie bardziej funkcjonalnego interfejsu. W przypadku „New Super Mario Bros” na górnym ekranie znajduje się widok rozgrywki, natomiast informacje o wyniku czy mapa poziomu znajdują się na dolnym ekranie. Widok obu ekranów został przedstawiony na rysunku 3.3. Wysoka przystępność spowodowała że gra zdobyła dużą popularność i sprzedała ponad 30 mln kopii ustanawiając jednocześnie rekord dla najlepiej sprzedającej się gry dla platformy Nintendo DS. Gra otrzymała od krytyków i graczy bardzo dobre recenzje średnio na poziomie 8.5/10, natomiast na serwisie Metacritic gra posiada średnią 94 punktów na 100.

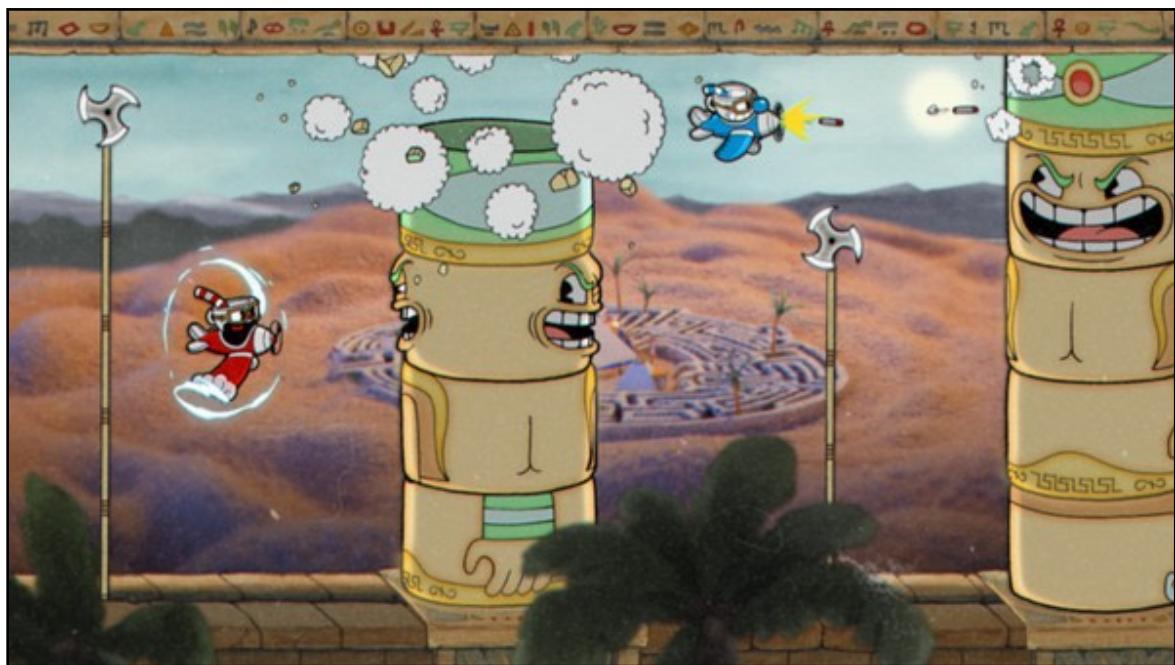


Rys. 3.3. - Zrzut z ekranów z gry „New Super Mario Bros”

Źródło: [https://mario.wiki.gallery/images/d/d3/  
NewMarioFlagpole.png](https://mario.wiki.gallery/images/d/d3/NewMarioFlagpole.png)

### 3.3.2. Cuphead

„Cuphead” to stworzona oraz wydana w 2017 roku przez Studio MDHR gra platformowa na komputery osobiste oraz na konsole z rodziny Xbox One oraz Playstation 4. Gra posiada unikatową oprawę graficzną (rys. 3.4.) i dźwiękową, która nawiązuje stylem do starych animacji z lat 30, np pierwszych animacji ze studia Disney. Postać gracza-tytułowy „Cuphead” na każdym poziomie posiada 3 życia, a same poziomy opierają się w dużej mierze na walkach ze specjalnymi silnymi przeciwnikami - „bossami”. Po śmierci pokazany jest postęp w misji i gracz zaczyna planszę od początku. Gra zawiera lokalny tryb kooperacji dla dwóch graczy. Gra zebrała szereg bardzo dobrych recenzji ze średnią 9/10, ponadto otrzymała wiele nagród jak i nominacji, w tym np. główną nagrodę w kategorii „The Best Independent Game” na gali The Game Awards w 2017 roku [3]. Do roku 2020 gra została sprzedana w ilości ponad 6 mln egzemplarzy.



Rys. 3.4. - Zrzut ekranu z gry „Cuphead”  
Źródło: <https://store.steampowered.com/app/268910/Cuphead/>

## **4. Opis wymagań systemu**

Funkcje w grze zostały podzielone ze względu na występowanie - menu, ustawienia, podczas trwania rozgrywki. Każda funkcja definiuje możliwe działanie użytkownika w danej sytuacji (Wymagania funkcjonalne).

Wymagania systemu zawierają informacje o zalecanym sprzęcie oraz oczekiwanyym interfejsie aplikacji, tak aby była przejrzysta dla użytkownika (wymagania niefunkcjonalne).

### **4.1. Wymagania funkcjonalne**

W sekcji menu:

- ▶ W celu umożliwienia spersonalizowanej rozgrywki, użytkownik ma możliwość rozpoczęcia nowej gry na jednym z profilów użytkownika.
- ▶ W celu kontynuowania rozgrywki, użytkownik ma możliwość wczytania zapisanej gry na jednym z profilów użytkownika.
- ▶ Aby rozpocząć rozgrywkę na konkretnym poziomie, użytkownik może skorzystać z opcji wyboru etapu, umożliwiającej mu rozpoczęcie gry od wybranego poziomu.
- ▶ W celu dostosowania środowiska do użytkownika, ma on możliwość zmiany ustawień.
- ▶ W celu zakończenia działania aplikacji, użytkownik ma możliwość wyjścia z gry.

W sekcji ustawień:

- ▶ W celu personalizacji poziomu głośności, użytkownik ma możliwość dostosowania jej do własnych potrzeb.
- ▶ W celu dostosowania wyświetlania do warunków oświetlenia, użytkownik ma możliwość zmiany jasności obrazu
- ▶ W celu ustawienia trybu wyświetlania według preferencji, użytkownik może go zmienić na pełnoekranowy lub okienkowy.
- ▶ W celu dostosowania jakości grafiki do możliwości sprzętowych, użytkownik ma możliwość zmiany rozdzielczości wyświetlania
- ▶ W celu dostosowania konfiguracji klawiszy, użytkownik ma możliwość zmiany przypisanych klawiszy
- ▶ W celu sprawdzenia aktualnej konfiguracji klawiszy, użytkownik ma możliwość sprawdzenia przypisanych klawiszy

Podczas trwania rozgrywki:

- ▶ W celu umożliwienia przemieszczenia postaci użytkownik ma możliwość poruszania nią za pomocą odpowiednich przycisków.
- ▶ W celu zapewnienia użytkownikowi informacji o aktualnym poziomie zdrowia postaci, system powinien wyświetlać liczbę życia.
- ▶ W celu umożliwienia interakcji z otoczeniem, system powinien reagować na odpowiednie akcje użytkownika.

- ▶ W celu umożliwienia zapisu stanu gry i późniejsze wznowienie rozgrywki, system powinien oferować możliwość zapisu gry przy punkcie kontrolnym.
- ▶ W celu zapewnienia kontroli nad rozgrywką, użytkownik ma możliwość wyjścia do menu.
- ▶ W celu dostosowania środowiska do użytkownika, ma on możliwość zmiany ustawień.
- ▶ W celu umożliwienia użytkownikowi tymczasowego przerwania rozgrywki, system powinien dostarczać funkcję wstrzymania gry.
- ▶ W celu umożliwienia użytkownikowi dynamicznego uczestnictwa w grze, system powinien dostarczać możliwość wykonywania akcji tj.: skok, atak, szarża, atak alternatywny.

## **4.2. Wymagania niefunkcjonalne**

Wymagania sprzętowe:

- ▶ Komputer z zainstalowanym systemem operacyjnym Windows
- ▶ Podłączone urządzenia peryferyjne
- ▶ Obsługa klawiatury

Wymagania dot. aplikacji:

- ▶ Przejrzysty interfejs użytkownika
- ▶ Intuicyjne sterowanie
- ▶ Intuicyjna budowa poziomów
- ▶ Zunifikowany wygląd czcionki
- ▶ Zaprogramowana w środowisku Unity

## **4.3 Definiowanie wymagań niefunkcjonalnych**

### ***4.3.1. Wymagania systemowe:***

- Komputer z oprogramowaniem systemowym Windows - aplikacja została zaprojektowana na komputery z systemem operacyjnym Windows. Wymaganie to gwarantuje, że aplikacja będzie działać zgodnie z oczekiwaniami na danym systemie operacyjnym.
- Obsługa klawiatury i myszki - aplikacja została zaprojektowana z myślą o wykorzystaniu klawiatury i myszki jako domyślnych urządzeń wejściowych.

### ***4.3.2. Wymagania dot. Aplikacji***

Wymagania niefunkcjonalne zostały opracowane na podstawie artykułu [4].

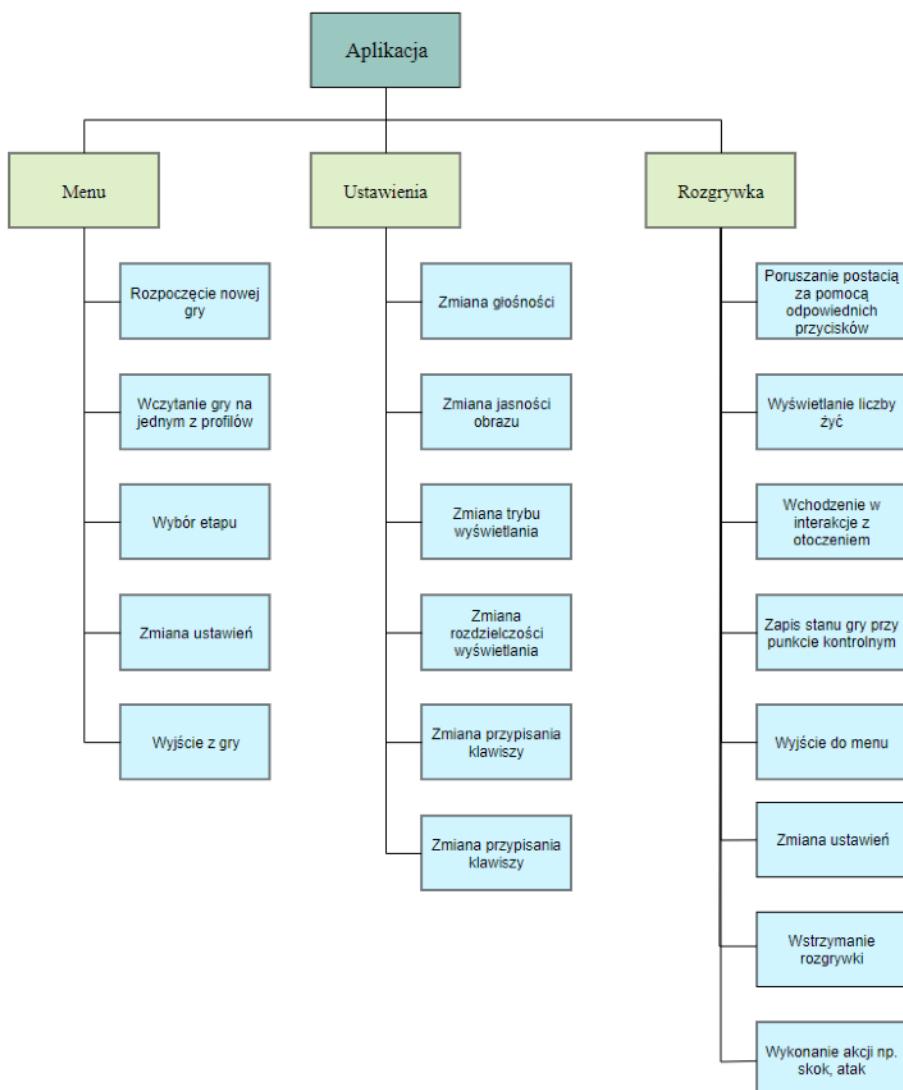
- Przejrzysty interfejs użytkownika - aplikacja zawiera łatwy i intuicyjny w obsłudze interfejs. Umożliwia on użytkownikowi przystępную nawigację oraz znalezienie potrzebnych funkcji w krótkim czasie.

- Intuicyjne sterowanie - program jako domyślny schemat sterowania posiada najpopularniejszy wśród gier z gatunku, układ przypisanych klawiszy „WSAD”.
- Zrozumiała konstrukcja poziomów - poziomy w grze posiadają logiczną, łatwą do zrozumienia przez gracza budowę, w której łatwo odnaleźć cel i go osiągnąć.
- Zunifikowany wygląd czcionki - aplikacja zapewnia spójność wyglądu czcionki we wszystkich elementach interfejsu.

#### 4.4 Tworzenie drzewa funkcji

Drzewo funkcji (rys. 4.1.) jest to graficzna wizualizacja hierarchii wymagań funkcjonalnych. Funkcjonalności zostały podzielone na trzy podkategorie:

- ▶ Menu – funkcjonalności dostępne z poziomu menu.
- ▶ Ustawienia – funkcjonalności dostępne z poziomu menu ustawień.
- ▶ Rozgrywka – funkcjonalności dostępne podczas rozgrywki.



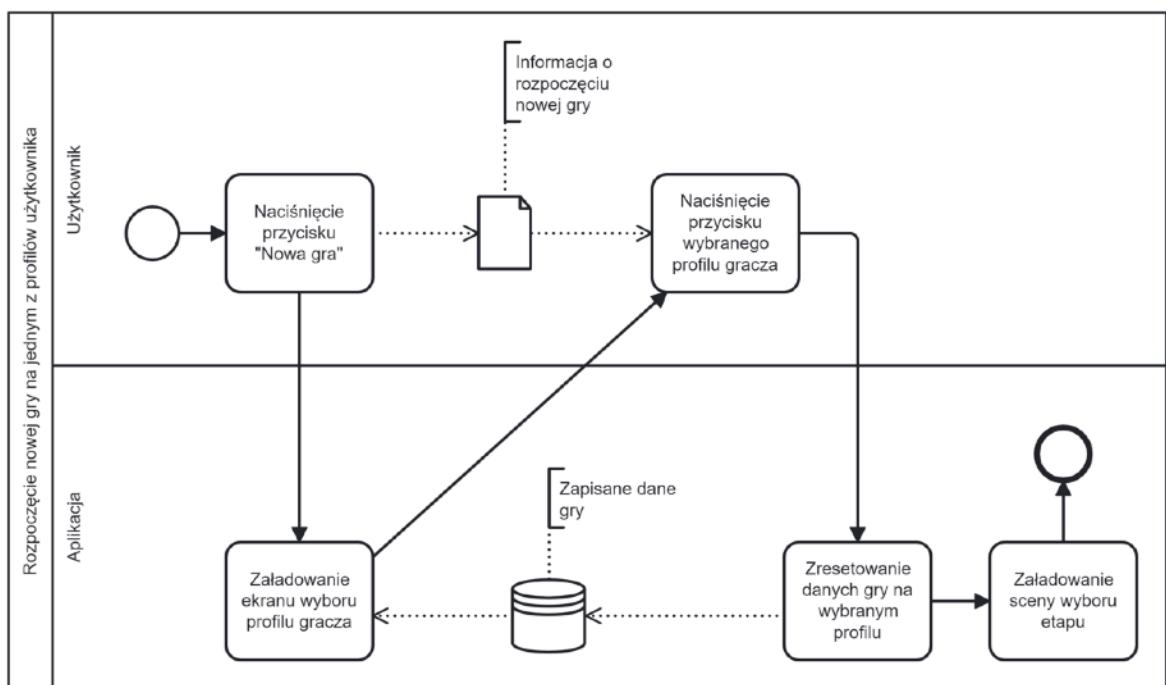
Rys. 4.1. Drzewo funkcji

## 5. Modelowanie wymagań funkcjonalnych

### 5.1. Tworzenie diagramów BPMN

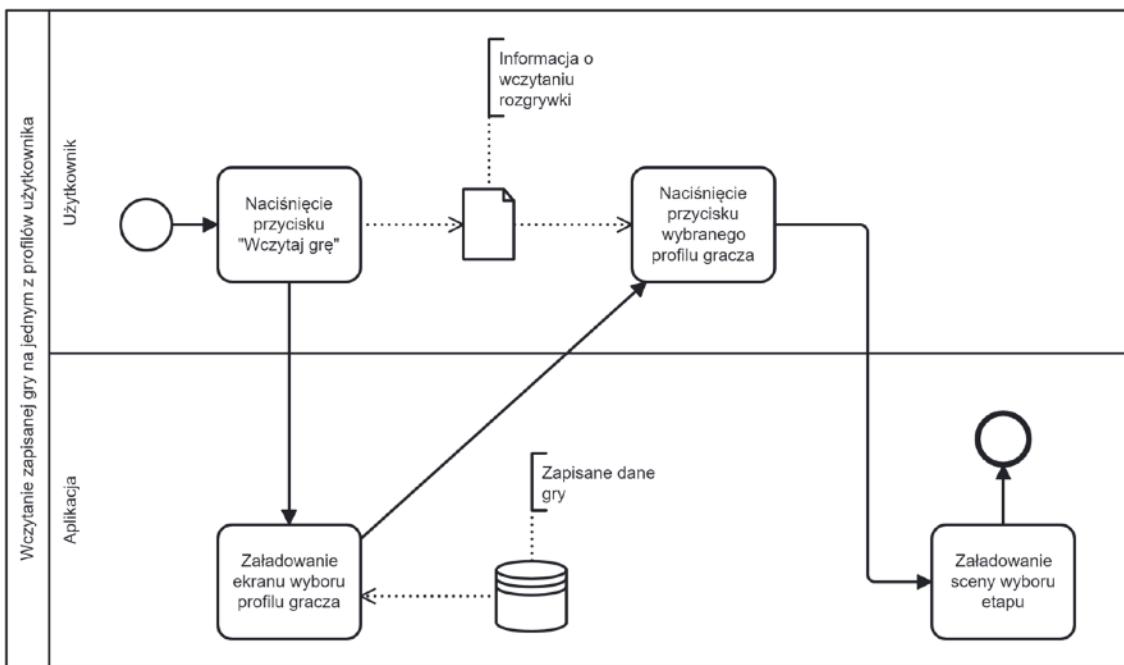
BPMN (Business Process Model and Notation) jest powszechnie stosowaną notacją graficzną, służącą do opisania funkcjonalności aplikacji. Poniższe diagramy BPMN zostały utworzone za pomocą programu Camunda Modeler, który udostępnia niezbędne narzędzia oraz elementy do utworzenia właściwych diagramów.

Poniższy diagram BPMN (rys. 5.1.) przedstawia funkcjonalność rozpoczęcia nowej gry. W momencie naciśnięcia przycisku „Nowa gra” przez użytkownika, zostanie przesłana informacja o konieczności zresetowania danych gry na wybranym przez niego profilu. Zapisane dane przechowywane są w pliku JSON (rozdział 6, podpunkt 1).



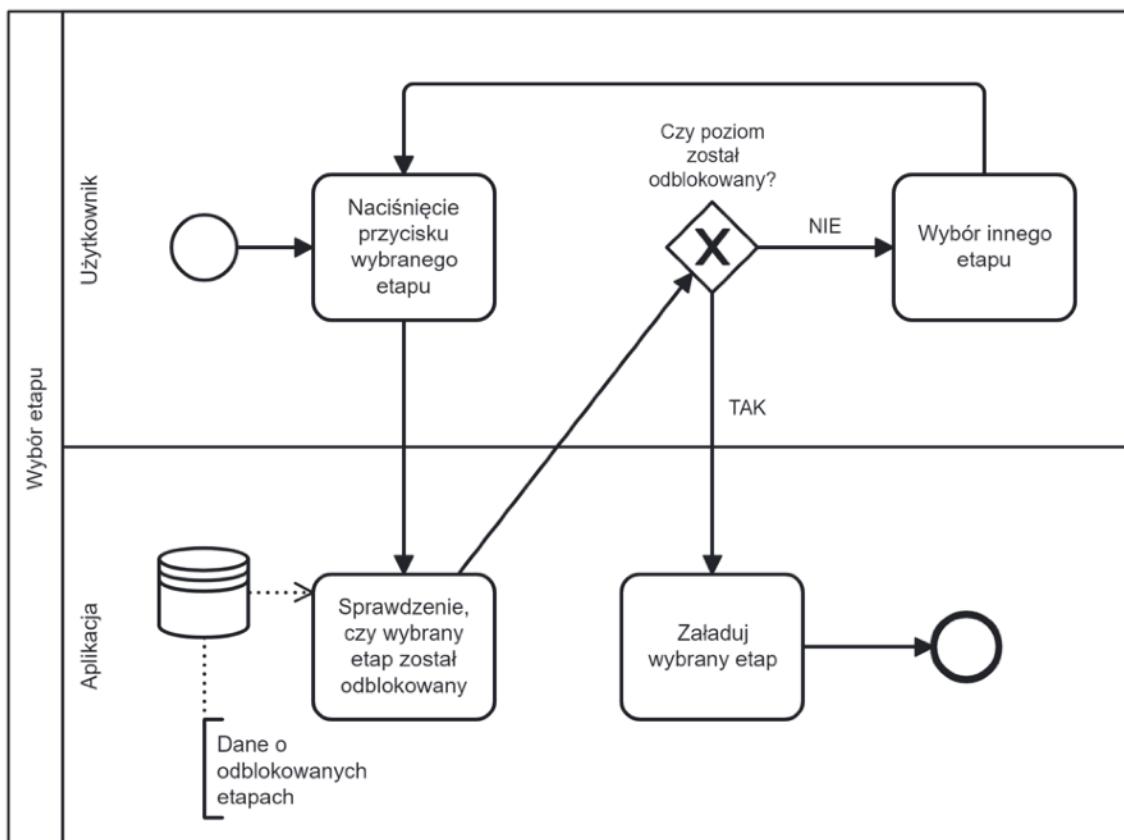
Rys. 5.1. Diagram funkcji rozpoczęcia nowej gry na jednym z profiliów użytkownika

Wczytanie zapisanej gry przez użytkownika przebiega analogicznie do rozpoczęcia „Nowej gry”. Zasadniczą różnicą jest fakt, że dane gry na wybranym profilu nie zostaną zresetowane. Jeżeli na profilu gracza nie znajdują się żadne zapisane dane, przycisk wyboru tego profilu będzie nieaktywny oraz wyświetli się na nim tekst „Profil jest pusty”. Diagram opisujący przebieg wczytania gry znajduje się na rys. 5.2.

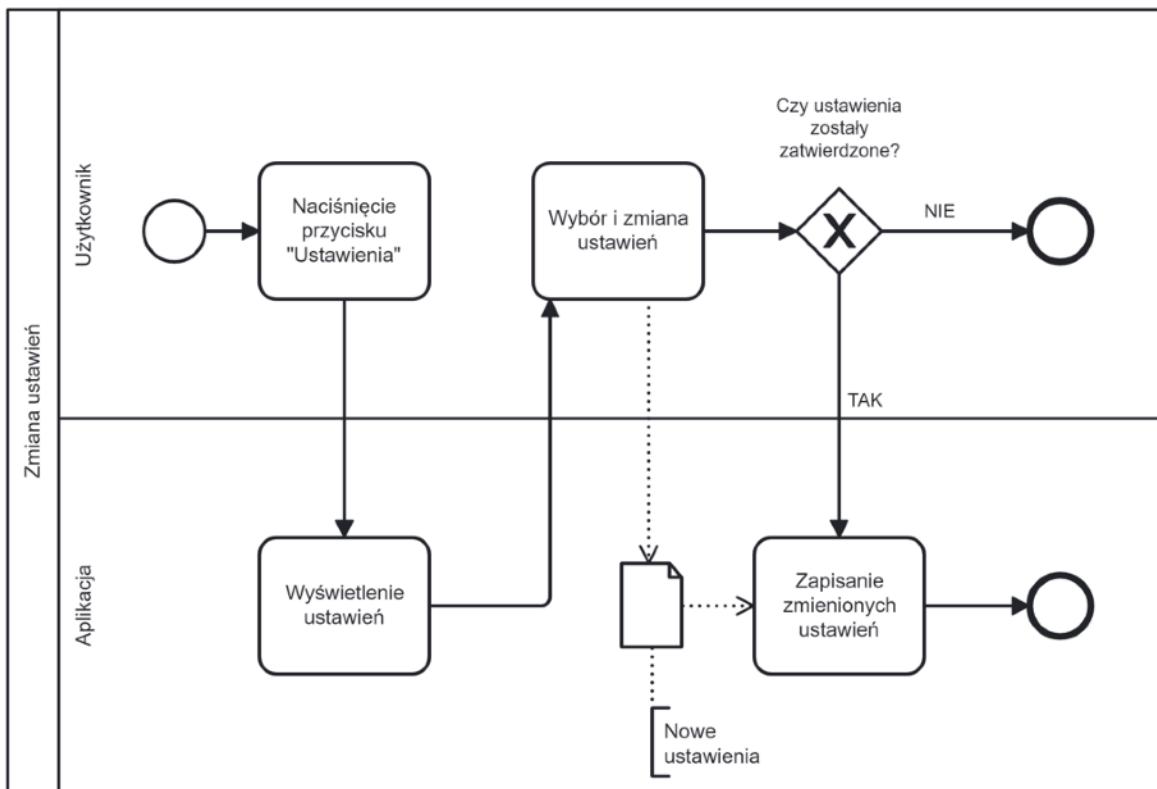


Rys. 5.2. Diagram funkcií wczytania zapisanej gry na jednym z profiliów użytkownika

Na scenie wyboru poziomu znajduje się pięć przycisków do wyboru etapu (rozdział 16, podpunkt 3). Jeżeli użytkownik rozpocznie „Nową grę”, wtedy dostępny jest tylko pierwszy etap. Ukończenie każdego z etapów, odblokowuje kolejny. Diagram przedstawiono na rys. 5.3.

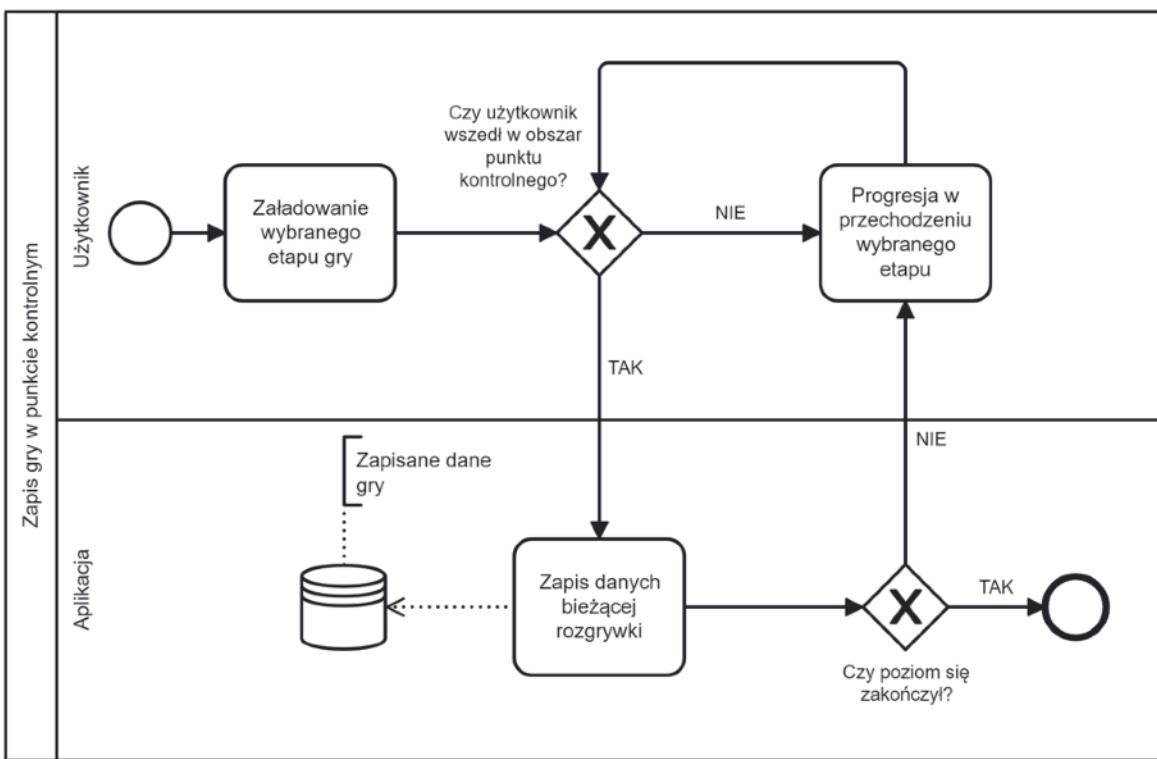


Użytkownik ma możliwość dostosowywać ustawienia według swojej preferencji (diagram przedstawiono na rys. 5.4.). Ustawienia dzielą się na „ogólne” oraz „sterowanie”. W obu przypadkach użytkownik musi zatwierdzić zmiany opcji, aby zostały one poprawnie zapisane.



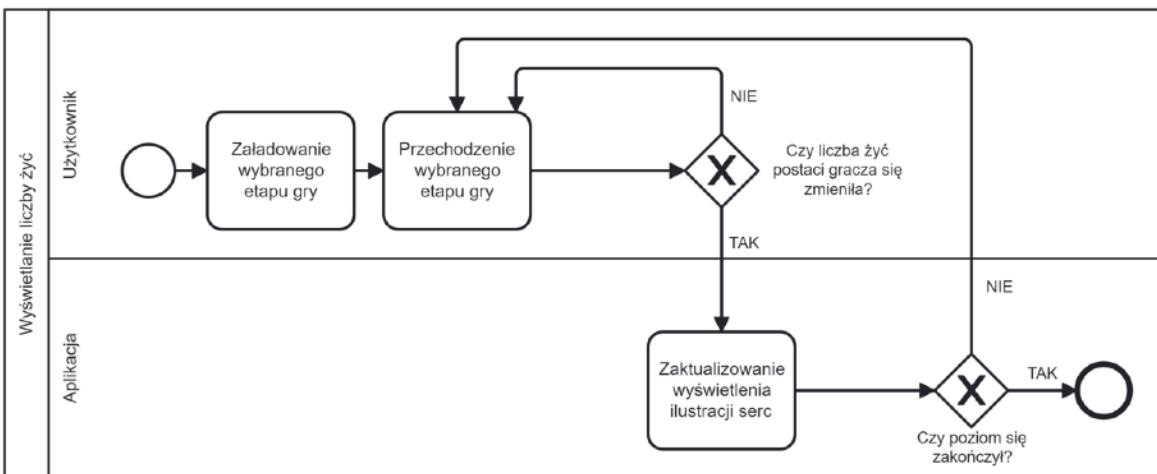
Rys. 5.4. Diagram funkcji obsługującej zmiany ustawień

Po załadowaniu wybranego przez użytkownika etapu, automatycznie następuje zapis gry. Kolejne punkty zapisu znajdują się przy określonych obiektach z nałożonym skryptem, sprawdzającym czy użytkownik wszedł w obszar danego obiektu. Każdy z punktów kontrolnych może być wykorzystany tylko raz, więc użytkownik nie może zapisać gry cofając się do wcześniejszego punktu. Diagram BPMN opisujący zapis gry został przedstawiony na rys. 5.5.



Rys. 5.5. Diagram funkcyjny obsługujący zapis gry w punktach kontrolnych

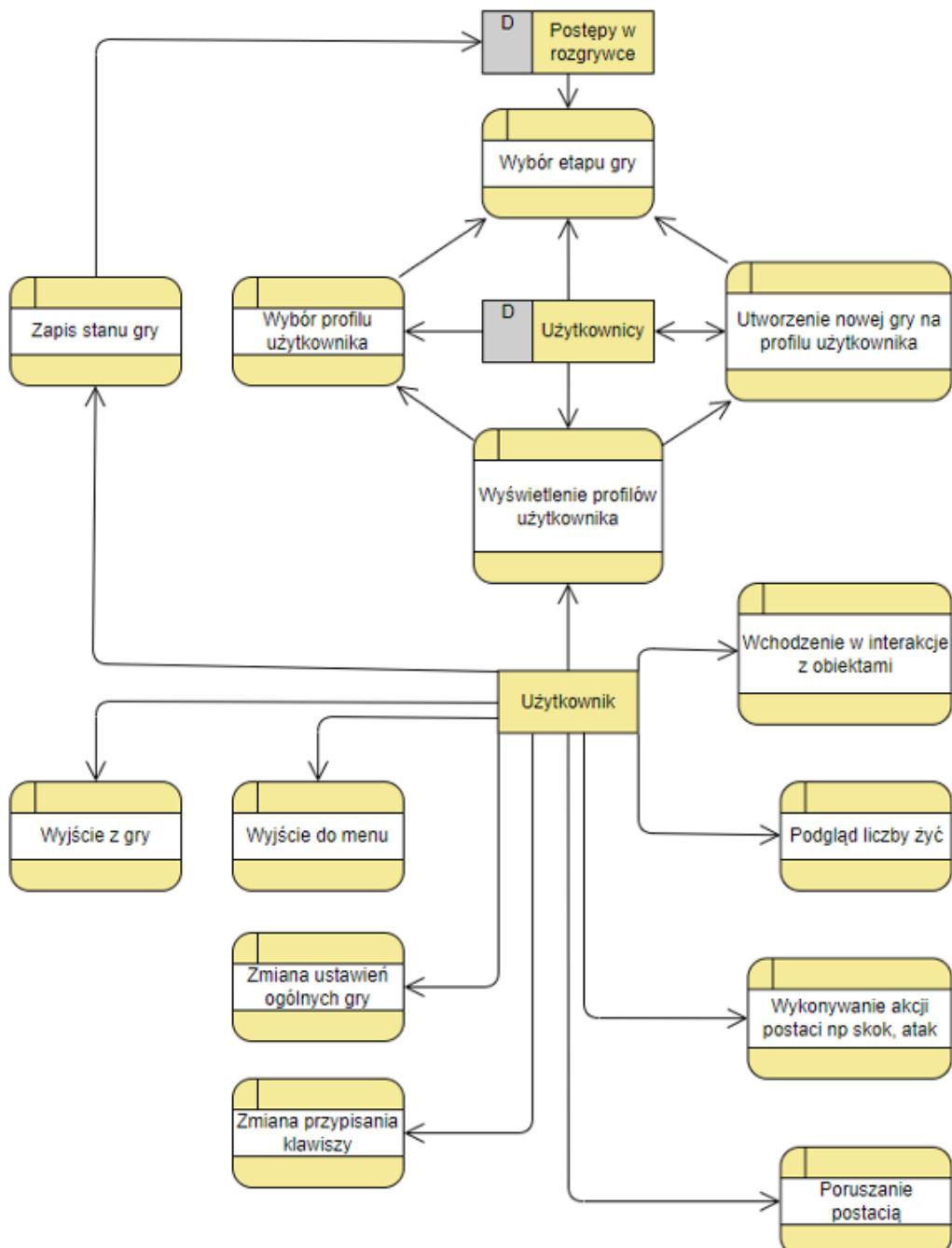
Podczas rozgrywki, na ekranie użytkownika wyświetlają się m.in. ilustracje serc, które symbolizują aktualny poziom zdrowia głównej postaci. Diagram przedstawiony na rys. 5.6. opisuje funkcjonalność wyświetlania aktualnej liczby życia. Dokładny opis systemu życie zostało zawarty w ramach opisu skryptów, które go obsługują (rozdział 13, podpunkt 6).



Rys. 5.6. Diagram funkcyjny obsługujący wyświetlanie liczby życia podczas rozgrywki

## **5.2 Tworzenie diagramu DFD**

Rys. 5.7. przedstawia diagram DFD (Data Flow Diagram) odpowiedzialny za wizualizację przepływu danych w zaprojektowanej aplikacji. Do zaprojektowania aplikacji wykorzystano dwa „data store” w celu przechowania danych użytkowników oraz ich postępów w rozgrywce. Dzięki zastosowaniu „data store” - „Postępy w rozgrywce” możliwe jest zablokowanie niedostępnych jeszcze etapów dla użytkownika. Etapy odblokowują się wraz z postępem w grze. Pozostałe funkcjonalności nie wymagają zapisu danych dlatego nie wymagają utworzenia „data store”.



Rys.5.7. Diagram DFD aplikacji

### 5.3 Definiowanie wymagań funkcjonalnych

Opisy procesów zostały zawarte w tabelach 5.1. – 5.9.

Tabela 5.1. Rozpoczęcie nowej gry na jednym z profilów użytkownika.

Nazwa funkcji (procesu)	Nowa gra
<b>Opis procesu</b>	Funkcja umożliwia utworzenie nowej gry w jednym z trzech dostępnych profili użytkownika.
<b>Wynik Działania</b>	Użytkownik zostaje przekierowany na scenę wyboru etapu.
<b>Uwagi</b>	- Jeżeli zostanie wybrany zapis z istniejącym już profilem, ten zostanie nadpisany.

Tabela 5.2. Wczytanie zapisanej gry na jednym z profili użytkownika.

Nazwa funkcji (procesu)	Wybór etapu
<b>Opis procesu</b>	Funkcja umożliwia uruchomienie wybranego etapu gry.
<b>Wynik Działania</b>	Użytkownik może zagrać w wybrany przez siebie etap.
<b>Uwagi</b>	- Użytkownik może jedynie wybrać dostępne etapy: etapy ukończone lub etapy bieżące.

Tabela 5.3. Wybór etapu

Nazwa funkcji (procesu)	Wybór etapu
<b>Opis procesu</b>	Funkcja umożliwia uruchomienie wybranego etapu gry.
<b>Wynik Działania</b>	Użytkownik może zagrać w wybrany przez siebie etap.
<b>Uwagi</b>	- Użytkownik może jedynie wybrać dostępne etapy - etapy ukończone lub etapy bieżące.

Tabela 5.4. Ustawienia Ogólne

Nazwa funkcji (procesu)	Ustawienia ogólne
<b>Opis procesu</b>	Funkcja umożliwia dostosowanie opcji gry do użytkownika.
<b>Wynik Działania</b>	Użytkownik zmienia wybrane ustawienia ogólne takie jak jasność, głośność, tryb wyświetlania, jakość grafiki.
<b>Uwagi</b>	- Użytkownik musi zatwierdzić zmiany.

Tabela 5.5. Zmiana ustawień sterowania

Nazwa funkcji (procesu)	
<b>Opis procesu</b>	Funkcja umożliwia dostosowanie przypisania klawiszy do akcji wykonywanych w grze.
<b>Dane wejściowe</b>	Kod wciskniętego klawisza.
<b>Źródło danych</b>	Klawiatura oraz mysz użytkownika.
<b>Wynik działania</b>	Użytkownik zmienia przypisanie klawiszy do wybranych akcji.
<b>Uwagi</b>	- Użytkownik musi zatwierdzić zmiany.

Tabela 5.6. Interakcja użytkownika z otoczeniem

Nazwa funkcji (procesu)	Interakcja użytkownika
<b>Opis procesu</b>	Funkcja umożliwia wchodzenie w interakcje z obiekta mi.
<b>Wynik Działania</b>	Użytkownik po wejściu w kolizję wraz z obiektem, wchodzi z nim w interakcje.
<b>Uwagi</b>	- Obiekt, z którym gracz wchodzi w kolizję musi być obiektem interaktywnym. - Użytkownik po interakcji z przeciwnikiem otrzymuje obrażenia. - Użytkownik po interakcji z obiektem reprezentującym punkt zdobywa punkt.

Tabela 5.7. Zapis stanu gry przy punkcie kontrolnym

Nazwa funkcji (procesu)	Zapis
<b>Opis procesu</b>	Funkcja umożliwia zapisanie stanu gry.
<b>Wynik Działania</b>	Użytkownik po wejściu w interakcję z punktem kontrolnym zapisuje postęp w rozgrywce.
<b>Uwagi</b>	- Punkt zapisu będzie wyróżniającym i powtarzanym obiektem, z którym trzeba wejść w interakcję. - Liczba i rozmieszczenie punktów kontrolnych będzie zależna od długości etapu. - Po ukończeniu etapu następuje automatyczny zapis postępów. - Przy interakcji użytkownik zostaje wyleczony oraz zapisana zostaje pozycja odrodzenia.

Tabela 5.8. Wstrzymanie gry

Nazwa funkcji (procesu)	Wstrzymanie gry
<b>Opis procesu</b>	Funkcja umożliwia wstrzymanie gry.
<b>Dane wejściowe</b>	Przycisk „Esc”.
<b>Źródło danych</b>	Klawiatura użytkownika.
<b>Wynik działania</b>	Po naciśnięciu przycisku „Esc” gra zostaje wstrzymana i zostaje wyświetlony panel menu pauzy.
<b>Uwagi</b>	Ponowne naciśnięcie przycisku „Esc” wznowia rozgrywkę.

Tabela 5.9. Wykonanie akcji tj: skok, bieg, atak, szarża, atak alternatywny

Nazwa funkcji (procesu)	Akcje użytkownika
<b>Opis procesu</b>	Funkcja umożliwia wykonywanie akcji podczas gry
<b>Dane wejściowe</b>	Przyciski przypisane do danych akcji.
<b>Źródło danych</b>	Klawiatura i mysz użytkownika.
<b>Wynik działania</b>	Po naciśnięciu odpowiedniego przycisku zostanie wywołana odpowiedzialna za niego akcja.
<b>Uwagi</b>	- Niektóre akcje wymagają dodatkowych warunków do wykonania czyt. aby wykonać atak alternatywny należy znajdować się na ziemi, akcja skoku może zostać wykonana dwukrotnie aż do momentu ponownego znalezienia się na ziemi.

## 6. Przechowywanie danych gry

### 6.1. Pliki JSON

W celu zapisywania danych z postępu rozgrywki, zastosowano konwersje danych do typu JSON. Takie podejście umożliwia przechowywanie danych gry z poszczególnych profili graczy w osobnych folderach w pliku "data.json". Na listingu 6.1. przedstawiono dane przechowywane w pliku tj.: liczba życia, pozycja gracza, zebrane punkty, czy liczba śmierci. Przy wczytaniu rozgrywki, dane są deserializowane z pliku JSON za pomocą biblioteki "JsonUtility".

```
"health": 6.0,  
"playerPosition": {  
    "x": 164.83567810058595,  
    "y": 0.06238079071044922,  
    "z": 0.0  
},  
"pointsCollected": 0,  
"deathCounter": 0
```

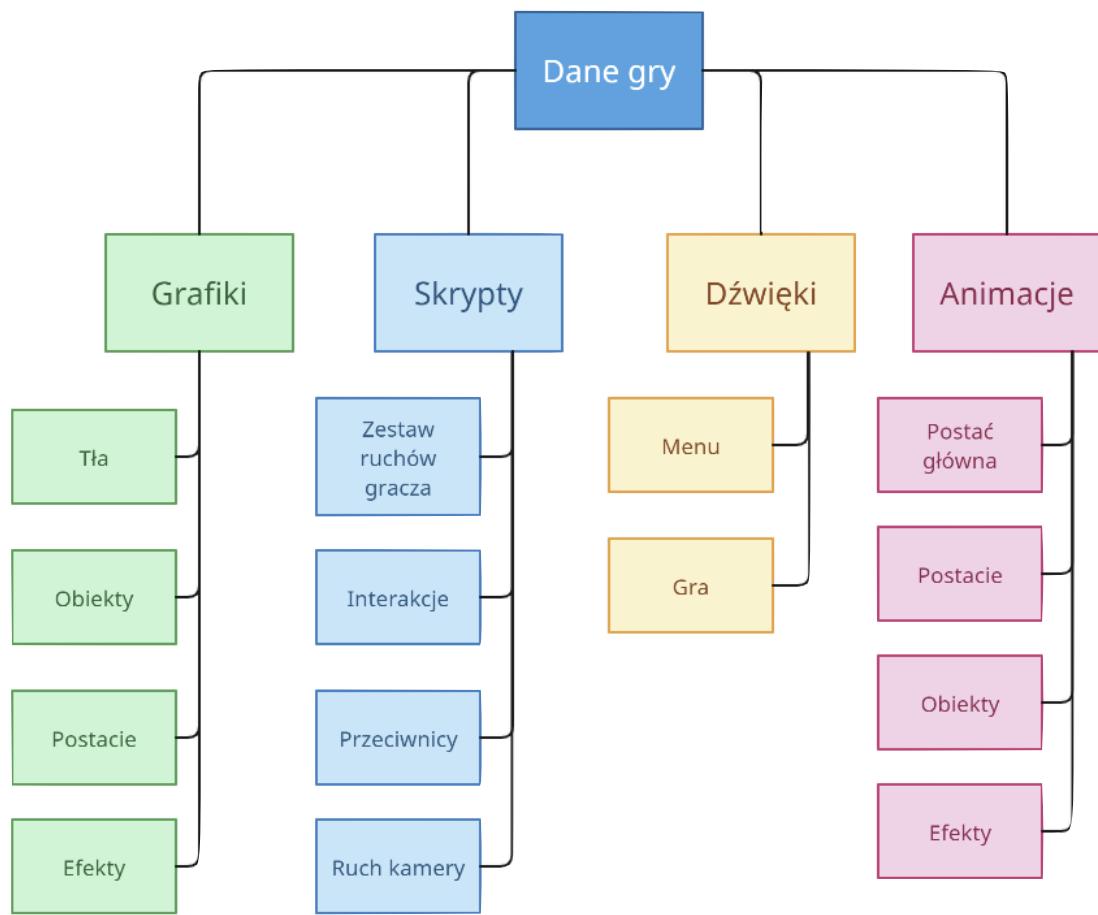
Listing 6.1. Przykładowa zawartość pliku "data.json"

### 6.2. Klasa "PlayerPrefs"

"PlayerPrefs" jest to klasa domyślnie dostępna w środowisku Unity [5]. Odpowiada ona za przechowywanie danych między różnymi sesjami gry. Zastosowanie "PlayerPrefs" jest wygodnym sposobem na zapisywanie i odczytywanie danych między scenami gry. Niezależnie od przechowywania danych, "PlayerPrefs" może też przechowywać wartość logiczną, co umożliwia zastosowanie logiki pomiędzy osobnymi scenami bez potrzeby tworzenia publicznych zmiennych. W ramach projektu, opisywana klasa została wykorzystana w celu przechowywania informacji o klawiszach przypisanych do akcji gracza oraz do zapisywania liczby zebranych punktów na danym poziomie gry.

### 6.3. Zasoby aplikacji

W grach tworzonych w Unity, dane gry, takie jak zasoby (np. grafiki, modele, animacje), są zazwyczaj przechowywane w folderach projektu. Folder "Assets" jest często używany do przechowywania wszystkich zasobów gry. W ramach tego folderu, można tworzyć podfoldery, aby uporządkować i kategoryzować różne rodzaje zasobów, co ułatwia zarządzanie nimi. Struktura przechowywania danych gry została przedstawiona na rys. 6.1.



Rys. 6.1. Schemat przechowywania danych gry

## **7. Opis elementów gry**

### **7.1. Opis głównej postaci**

#### **Opis fabularny:**

Podczas rozgrywki gracz wciela się w postać królowej warcabów, której celem jest zemsta za śmierć męża.

#### **Zakres ruchów:**

- Ruch – domyślnie "WSAD". Postać przemieszcza się w kierunku wybranym przez gracza. Na ruch mogą wpływać elementy otoczenia – struktura powierzchni, poruszające się platformy.
- Skok - domyślnie "spacja". Postać wykonuje podskok wzwyż lub wybranym przez gracza kierunku.
- Atak podstawowy – domyślnie "LPM". Postać wykonuje zamach berłem o krótkim zasięgu, zadającą obrażenia przeciwnikom.
- Atak alternatywny (wystrzał z berła) – domyślnie "PPM". Postać używa berła jako broni dystansowej, pocisk zostaje wystrzelony w kierunku wybranym przez gracza.
- Szarża – domyślnie naciśnięcie „Lewy Shift”. Postać szarżuje w wybranym przez gracza kierunku. Pęd może zostać przerwany przez trafienie na obiekt fizyczny.

### **7.2. Opis przeciwników podstawowych**

#### *7.2.1. Biały Pion*

#### **Opis Fabularny:**

Jest to podstawowy i najczęściej spotykany przeciwnik w armii Białego Króla Szachów.

#### **Zestaw ruchów:**

##### Przemieszczenie:

- Pion porusza się między wyznaczonymi punktami gdy dotrze do jednego z punktów zaczyna poruszać się w stronę kolejnego punktu.
- Gdy gracz znajdzie się w zasięgu wzroku przeciwnika ten zaczyna za nim podążać.

##### Podstawowy atak:

- Gdy gracz znajdzie się w zasięgu ataku przeciwnika ten wykonuje „Cios maczugą”

#### *7.2.2. Biały Skoczek*

#### **Opis Fabularny:**

Jest to kawaleria w armii Białego Króla Szachów.

**Zestaw ruchów:**

Przemieszczenie:

Skoczek przemieszcza się w określonym kierunku aż do momentu napotkania przeszkody, bądź braku podłożu wtedy też skoczek zmienia kierunek ruchu.

Podstawowy atak:

Gdy gracz znajdzie się w zasięgu ataku Skoczka ten wykonuje Skok w kierunku gracza.

### *7.2.3. Biały Goniec*

**Opis Fabularny:**

Jest to zwiadowca w armii Białego Króla Szachów.

**Zestaw ruchów:**

Przemieszczenie:

Goniec porusza się latając między dwoma punktami umieszczonymi po przekątnej.  
Po dotarciu do punktu zaczyna poruszać się w kierunku następnego punktu.

Podstawowy atak:

Goniec nie posiada własnego ataku, jedynie wejście z nim w kolizje zada obrażenia graczowi.

### *7.2.4. Biała Wieża*

**Opis Fabularny:**

Jest to fortyfikacja w armii Białego Króla Szachów.

**Zestaw ruchów:**

Przemieszczenie:

Biała wieża nie posiada możliwości przemieszczania się.

Podstawowy atak:

Jeżeli gracz znajdzie się w zasięgu ataku wieży ta wystrzeli strzałę w kierunku gracza. Pocisk zostaje zniszczony przy kontakcie z graczem, podłożem lub po upłynięciu określonego czasu. Kolejne pociski są wystrzeliwane w równych odstępach czasowych.

### *7.2.5. Czarny Pion*

**Opis Fabularny:**

Jest to piechur w armii Czarnego Króla Szachów. Posiada włócznię zwiększającą zasięg jego ataku względem odpowiednika z armii Białego Króla Szachów.

**Zestaw ruchów:**

Przemieszczenie:

1. Pion porusza się między wyznaczonymi punktami gdy dotrze do jednego z punktów zaczyna poruszać się w stronę kolejnego punktu.
2. Gdy gracz znajdzie się w zasięgu wzroku przeciwnika ten zaczyna za nim podążać.

Podstawowy atak:

Gdy gracz znajdzie się w zasięgu ataku przeciwnika ten wykonuje „Pchnięcie włócznią”.

#### *7.2.6. Czarny Skoczek*

**Opis Fabularny:**

Jest to kawaleria w armii Czarnego Króla Szachów. Posiada znacznie większy zasięg widzenia od jego odpowiednika z armii Białego Króla Szachów.

**Zestaw ruchów:**

Przemieszczenie:

Skoczek przemieszcza się w określonym kierunku aż do momentu napotkania przeszkody, bądź braku podłożu wtedy też skoczek zmienia kierunek ruchu.

Podstawowy atak:

Gdy gracz znajdzie się w zasięgu ataku Skoczka ten wykonuje Skok w kierunku gracza.

#### *7.2.7. Czarna Wieża*

**Opis Fabularny:**

Jest to fortyfikacja w armii Czarnego Króla Szachów.

**Zestaw ruchów:**

Przemieszczenie:

Czarna wieża nie posiada możliwości przemieszczania się.

Podstawowy atak:

Jeżeli gracz znajdzie się w zasięgu ataku wieży ta wystrzeli magiczny pocisk, który podąża za graczem. Pocisk zostaje zniszczony przy kontakcie z graczem, podłożem lub po upłynięciu określonego czasu. Kolejne pociski są wystrzeliwane w równych odstępach czasowych.

## **7.3. Opis przeciwników specjalnych**

### *7.3.1. Leniwy Naczelnik Więzienia*

#### **Opis fabularny:**

Jest to zarządca więzienia, w którym została osadzona bohaterka. Jako uzbrojenie posiada on szynkę, która jest traktowana jako maczuga.

#### **Zestaw ruchów:**

Faza 1:

Przemieszczenie:

Naczelnik porusza się idąc w stronę gracza.

Podstawowe ataki:

1. Cios szynką
2. Kopnięcie
3. Szybkie cięcie szynką
4. Cięcie od dołu szynką

Faza 2:

Przemieszczenie:

Naczelnik porusza się biegnąc w stronę gracza.

Podstawowe ataki:

Są to ataki z pierwszej fazy bossa jednak ich animacje są szybciej wykonywane.

Umiejętności specjalne:

1. Rzut szynką – Jeżeli gracz znajduje się w określonej odległości od Naczelnika, ten rzuca w jego stronę szynką.
2. Leczenie - Raz na walkę, gdy naczelnik osiągnie 25% swojego poziomu zdrowia, używa on umiejętności, która pozwala na powolną regenerację zdrowia (1/s), gdy zdrowie osiągnie wartość 50% bądź zostanie on zaatakowany podczas regeneracji umiejętność zostaje przerwana.

### *7.3.2. Rozjuszony Naczelnik Więzienia*

#### **Opis fabularny:**

Jest to wcześniej pokonany Leniwy Naczelnik Więzienia, który ruszył w pogон za Królową. Nie posiada on już uzbrojenia ani drugiej fazy.

#### **Zestaw ruchów:**

Przemieszczenie:

Naczelnik porusza się biegnąc w stronę gracza.

Podstawowe ataki:

1. Okrzyk bojowy – Naczelnik krzyczy odpychając gracza od siebie
2. Skok - Naczelnik skacze na niewielką wysokość, w momencie wyskoku gracza, po wylądowaniu tworzy on falę uderzeniową.
3. Duży skok - Naczelnik skacze na dużą wysokość, po wylądowaniu tworzy on falę uderzeniową.
4. Upadek - Naczelnik przewraca się przed graczem tworząc falę uderzeniową w momencie upadku.
5. Toczenie - Naczelnik obrót na powierzchni w kierunku gracza.

### 7.3.3. *Biały Król*

#### **Opis fabularny:**

Jest to cel zemsty Królowej Warcabów.

#### **Zestaw ruchów:**

Umiejętność Pasywna:

Podczas walki dodatkowym utrudnieniem są spadające kamienie.

Przemieszczenie:

Król porusza się przesuwając się po podłożu w stronę gracza.

Umiejętności specjalne:

1. Biały Król może przyzywać określone białe figury na pole walki, które pomogą mu w walce przeciwko graczowi.
2. Biały Król otacza się osłoną, która blokuje obrażenia zadawane przez gracza do momentu zniszczenia.

### 7.3.4. *Czarny Król*

#### **Opis fabularny:**

Jest to główny antagonist gry, który manipułował Królową Warcabów.

#### **Zestaw ruchów:**

Przemieszczenie:

Król porusza się przesuwając się po podłożu w stronę gracza.

Umiejętności specjalne:

1. Czarny Król może przyzywać określone czarne figury na pole walki, które pomogą mu w walce przeciwko graczowi.
2. Czarny Król tworzy dwa kolce, zadające obrażenia graczowi przy kontakcie.
3. Czarny Król tworzy piorun, zadający obrażenia graczowi przy kontakcie.

## **7.4. Opis Pułapek**

### *7.4.1. Kolce*

#### **Opis działania:**

Są to zaostrzone obiekty wystające z powierzchni podłogi lub ściany. Zadają obrażenia przy kontakcie z bohaterką.

### *7.4.2. Piła tartaku*

#### **Opis działania:**

Jest to ostrze poruszające się poziomo. Zadaje obrażenia bohaterce przy kontakcie z bohaterką.

## **8. Zarys fabuły**

### **8.1. Krótki opis fabularny**

Po przegranej wojnie z szachami i śmierci króla warcabów nastąpił pokój. Jednak „Królowa Warcabów” uwięziona w lochach planuje zemstę na mordercy jej męża.

### **8.2. Zarys świata**

Świat „Checkmate Highlands” odbudowuje się po licznych wojnach. Dzięki ciężkiej pracy „Białego Króla Szachów” kraj warcabów zaczyna ponownie stawać na nogi.

### **8.3. Prolog**

Od wielu wieków warcaby toczyły wojny z szachami, jednak w wielkiej bitwie poległ król warcabów, przez co kraj upadł, a władzę przejął „Biały Król Szachów”. Liczni mieszkańcy krainy warcabów rozpierzchli się po świecie i czekają na powrót „Królowej”, która została wtrącona do lochu, gdzie planuje swoją zemstę.

### **8.4. Akt I - Ucieczka**

Na najniższym piętrze lochu, gdzie znajdowała się „Królowa”, widok straży był nieczęsty. Pewnej nocy nikt nie pełnił warty, a „Królową” obudziła zakapturzona postać. Postać ta zaoferowała „Królowej” wolność w zamian za przysługę, o której miała się dowiedzieć niebawem. Nieznajomy wręczył „Królowej” berło i rozpłynął się w mrokach lochu. „Królowa” nie zastanawiając się dłużej, ruszyła w kierunku wyjścia omijając ciała wartowników. Jest świadoma, że aby uciec musi pokonać „Leniwego Naczelnika Więzienia”.

### **8.5. Akt II - Pościg**

„Królowa” po udanej ucieczce z więzienia nie może sobie pozwolić na chwilę wytchnienia, ponieważ jest ona poszukiwana listem gończym. Bohaterka zdaje sobie sprawę, że nie jest w stanie pokonać przeważających sił wroga, co nakłania ją do odnalezienia popleczników i zwiększenia swoich sił. Jednak najpierw musi znaleźć schronienie.

### **8.6. Akt III - Bezpieczna przystań**

Po odnalezieniu ukrytego miasteczka warcabów, królowa zdobywa informacje odnośnie sytuacji politycznej w „Highlands” oraz nakłania warcaby do powstania przeciwko „Białemu Królowi”. Wydostanie się z ukrytego miasta jednak nie jest takie proste, ponieważ wymaga przedostania się przez niebezpieczny las.

## **8.7. Akt IV - Zemsta**

„Królowa” dociera pod bramę zamku „Białego Króla Szachów”, ku jej zdziwieniu widzi sztandary białych i czarnych szachów. Rozpoczyna się oblężenie, podczas którego królowa przedziera się do sali tronowej, gdzie walczy przeciwko „Białemu Królowi Szachów”. Po pokonaniu przeciwnika przed królową pojawia się zakapturzona postać, która informuje ją o wypełnieniu przysługi, okazuje się że owa postać jest poplecznikiem „Czarnego Króla Szachów”.

## **8.8. Akt V - Gorzka prawda**

„Królowa zrozumiała”, że była jedynie pionkiem w intrydze uknutej przez „Czarnego Króla Szachów”. Dowiaduje się, że „Biały Król Szachów” nie był odpowiedzialny za śmierć jej męża, a po jego śmierci troszczył się o jej poddanych. To „Czarny Król Szachów” był mordercą jej męża oraz wyzwolicielem z lochu, zrobił to ponieważ pragnął śmierci „Białego Króla”, gdyż pożądał władzy absolutnej. W tym celu napuścił żądny zemsty bohaterkę na „Białego Króla”. Po odkryciu prawdy pełna smutku i złości „Królowa” wyrusza na bitwę przeciwko „Czarnemu Królowi”.

## **8.9. Epilog**

Wieść o śmierci „Czarnego Króla Szachów” rozniosła się po całym „Checkmate Highlands”. Ocalałe warcabы opuściły swoje kryjówki i powróciły do swoich domów. „Królowa Warcabów” rządziła zgodnym zjednoczonym ludem warcabów oraz szachów oddając przy tym honory zmarłemu „Królowi Białych Szachów”.

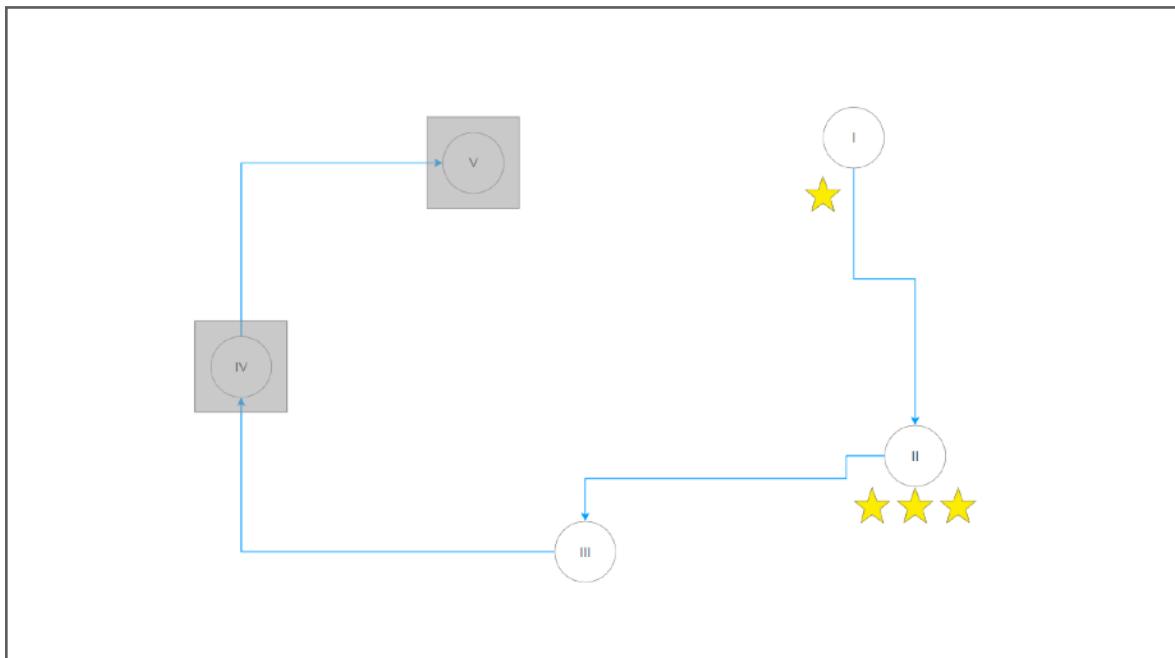
## 9. Projektowanie widoków aplikacji

Na rysunku 9.1. zaprezentowano widok menu głównego, jest on złożony z czterech podstawowych funkcji. Po naciśnięciu przycisku „Nowa gra” następuje przekierowanie do zakładki profiliów. W ten sam sposób działa przycisk „Wczytaj grę”. Przycisk „Ustawienia” przekierowuje użytkownika do panelu wyboru ustawień.



Rys. 9.1. Widok menu głównego

Rys 9.2. obrazuje wybór etapu gry. Przy ukończonych poziomach znajdują się gwiazdki reprezentujące liczbę zebranych punktów na danym poziomie. Poziomy zasłonięte szarym prostokątem są zablokowane dla gracza do momentu ukończenia poprzedniego etapu.



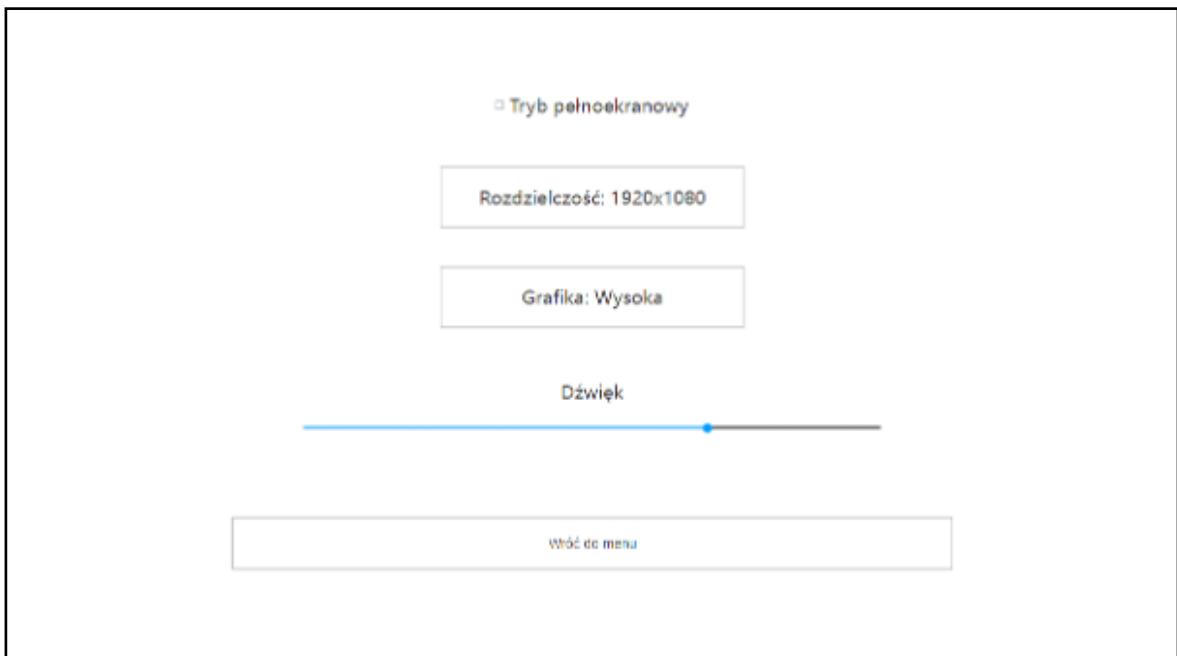
Rys. 9.2. Widok menu wyboru poziomu

Rys 9.3. przedstawia widok wyboru ustawień. Przycisk „Ogólne” przekierowuje użytkownika do panelu „Ustawień ogólnych”, natomiast przycisk „Sterowanie” przekierowuje do panelu odpowiedzialnego za przypisanie klawiszy.



Rys.9.3. Widok wyboru ustawień

Na rysunku 9.4. został zaprojektowany poglądowy wygląd panelu „Ustawień ogólnych”. Są to ustawienia dotyczące wyświetlania oraz głośności.



Rys. 9.4. Widok ustawień ogólnych

Rys. 9.5. przedstawia przykładowy panel „Przypisania klawiszy”, umożliwia on użytkownikowi przypisanie wybranych przez niego klawiszy do danej akcji.



Rys. 9.5. Widok ustawień sterowania

Na rys. 9.6. został zaprezentowany przykład interfejsu użytkownika oraz przykładowy fragment poziomu.



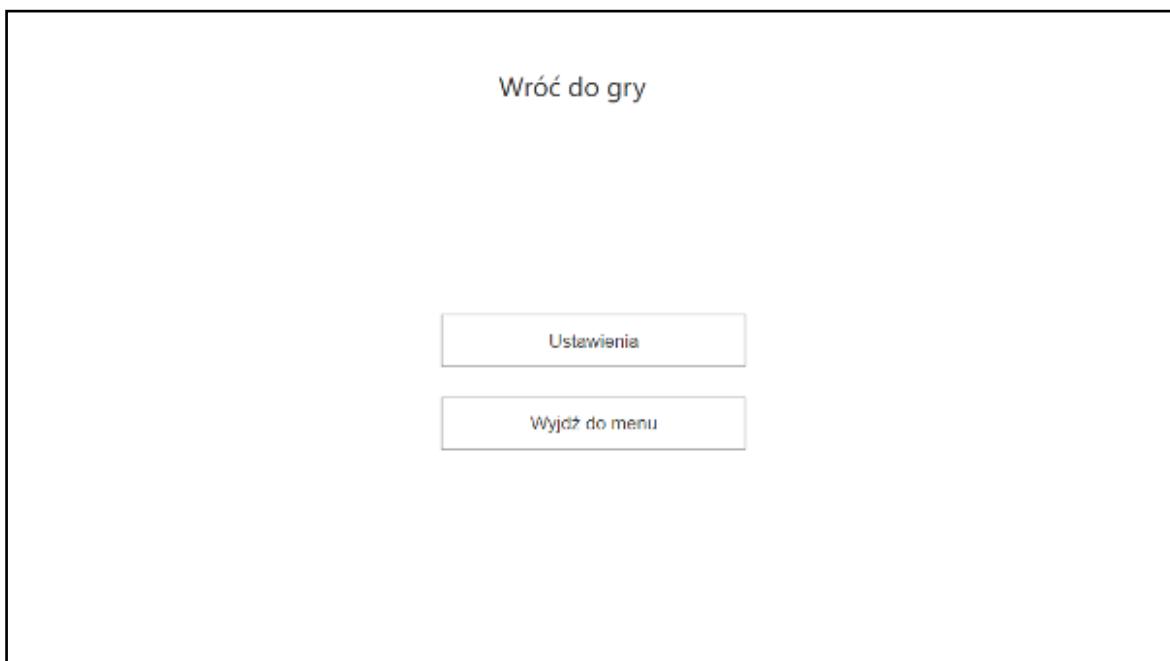
Rys. 9.6. Widok interfejsu użytkownika

Na rys. 9.7. zaprezentowano przykład walki z przeciwnikiem specjalnym. Na dolnej części ekranu zostaje wyświetlona wartość życia przeciwnika.



Rys. 9.7. Widok walki z przeciwnikiem specjalnym

Rys. 9.8. przedstawia widok ekranu pauzy, który jest uruchamiany po naciśnięciu przycisku „Esc” podczas rozgrywki.



Rys. 9.8. Widok ekranu pauzy

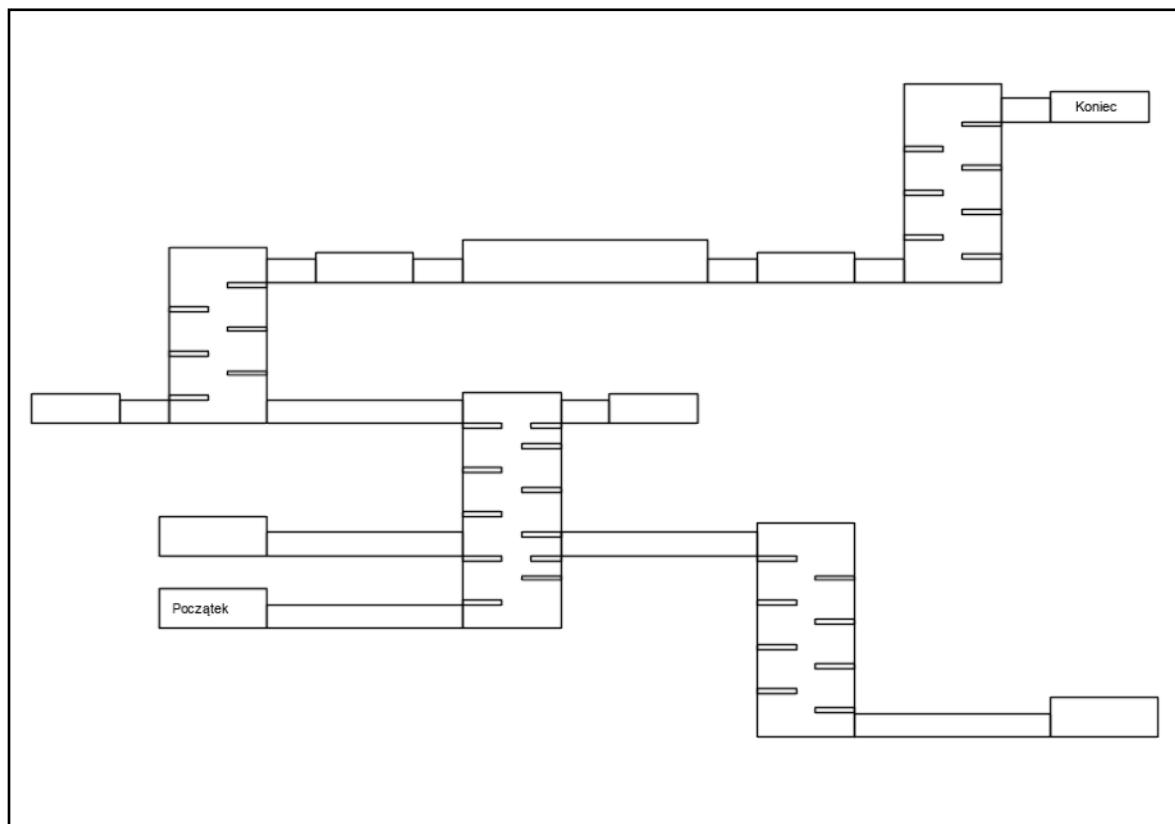
## 10. Projekt i implementacja poziomów

### 10.1. Tworzenie poziomów w Unity

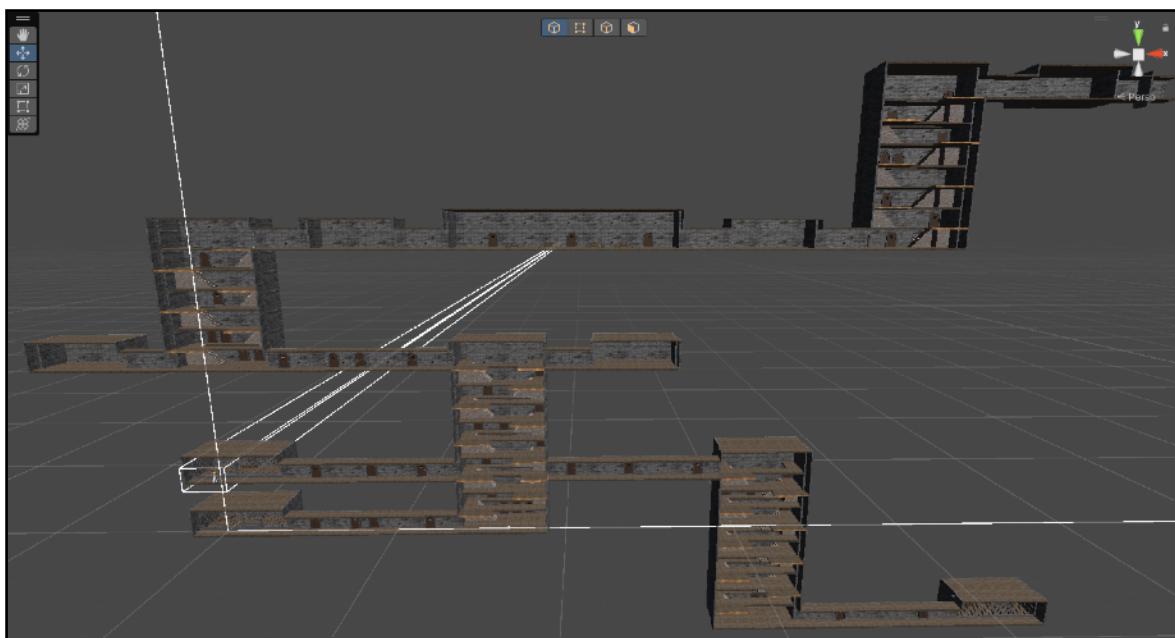
Dla każdego poziomu została stworzona oddzielna scena w projekcie Unity. Do tworzenia poziomów zostało wykorzystane narzędzie ProBuilder które jest zintegrowane z silnikiem Unity. Narzędzie to pozwala na tworzenie i edycję geometrii trójwymiarowej w czasie rzeczywistym. Poza możliwością edycji prostych brył geometrycznych, ProBuilder pozwala również na zaawansowaną edycję wierzchołków, krawędzi i ścian, a także na proceduralne generowanie złożonych kształtów takich jak: schody, łuki, drzwi itp. Bezpośrednia integracja z Unity umożliwia płynną współpracę z innymi elementami projektu, co wraz z intuicyjnym interfejsem sprawia, że ProBuilder jest niezastąpionym narzędziem zarówno do prototypowania scen, jak i tworzenia gotowych poziomów. Podczas tworzenia poziomu zastosowano zasady znajdujące się w artykule [6].

### 10.2. Poziom 1 - Więzienie

Jest to pierwszy poziom gry, na którym gracz poznaje sterowanie oraz podstawowe mechaniki gry. Celem "Królowej" jest znalezienie wyjścia z więzienia, błądząc po korytarzach i klatkach schodowych budynku. Po drodze, główna bohaterka napotyka przeciwników - strażników których musi pokonać. Poziom kończy się w momencie kiedy "Królowa" trafi do wyjścia. Projekt oraz implementacja poziomu 1 pokazana jest na rysunkach 10.1 oraz 10.2.



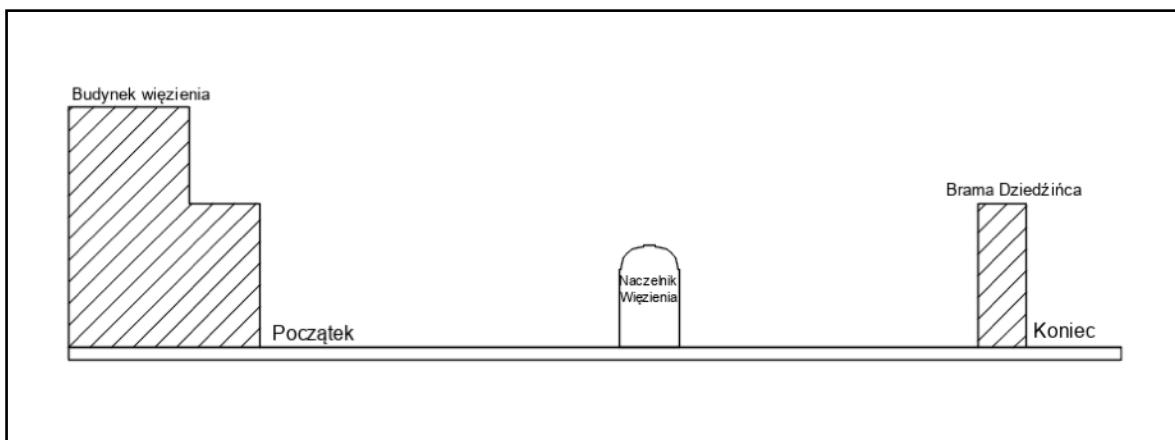
Rys. 10.1. Plan poziomu pierwszego



Rys. 10.2. Widok poziomu I w edytorze Unity

### 10.3. Poziom 1 - Walka z Naczelnikiem Więzienia

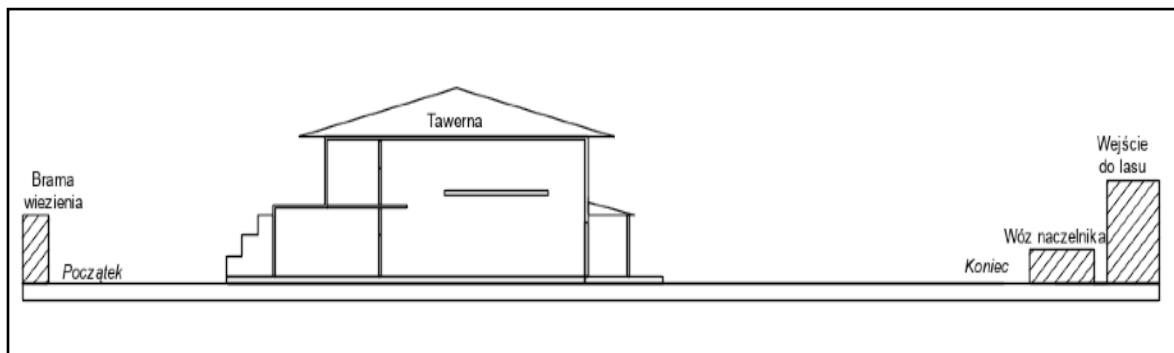
Po wyjściu z budynku więzienia, główna bohaterka trafia na dziedziniec gdzie staje do walki z pierwszym przeciwnikiem specjalnym: "Leniwym Naczelnikiem Więzienia". Po pokonaniu "Naczelnika", "Królowa" opuszcza dziedziniec przez główną bramę. Projekt poziomu widoczny jest na rysunku 10.3.



Rys. 10.3. Projekt poziomu 2

#### **10.4. Poziom 2 - Ucieczka przez wsie**

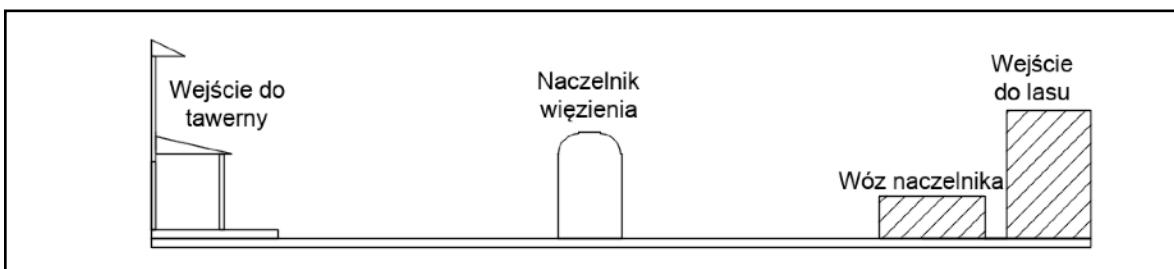
Jest to drugi poziom gry. Za bramami więzienia "Królowa" kontynuuje swoją ucieczkę biegając przed siebie przez okoliczne pola i łąki. Po pewnym czasie główna bohaterka dociera do tawerny, do której dostaje się przez balkon znajdujący się z tyłu i przekradając się przez nią kontynuuje swoją podróż. Po wyjściu z tawerny, królowa dociera do granicy lasu gdzie napotyka wóz. Projekt poziomu znajduje się na rysunku 10.4.



Rys. 10.4. - Projekt poziomu 2

#### **10.5. Poziom 2 - Ponowna walka z Naczelnikiem**

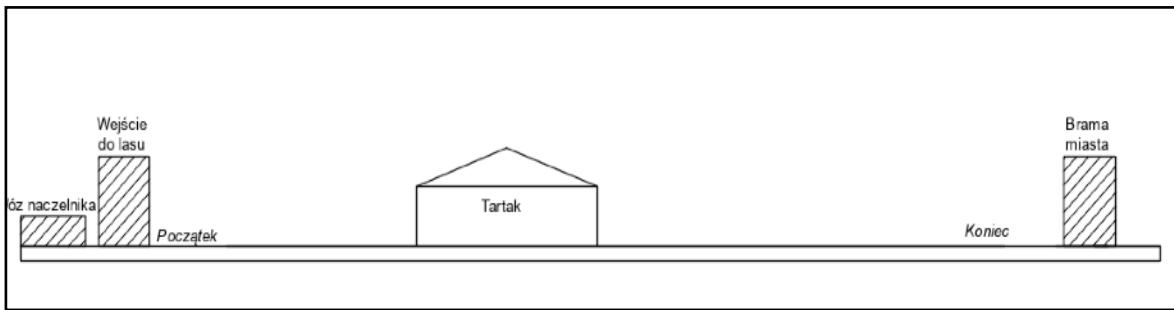
Po dotarciu do wozu okazuje się że przyjechał nim "Naczelnik Więzienia", który gonił bohaterkę od bram więzienia. Dochodzi do ponownej walki z "Naczelnikiem". Po pokonaniu przeciwnika, "Królowa" ucieka do lasu, a poziom zostaje ukończony. Projekt poziomu znajduje się na rysunku 10.5.



Rys. 10.5. Projekt poziomu walki po 2 poziomie

#### **10.6. Poziom 3 - Wędrówka przez las**

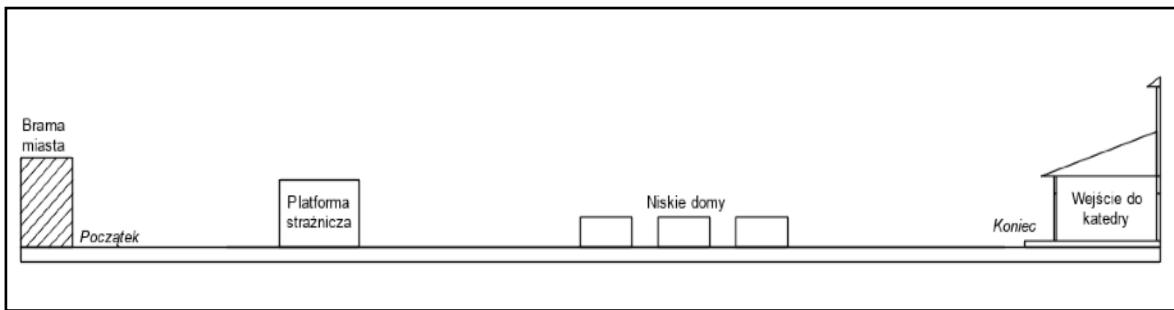
Jest to trzeci poziom gry (rys. 10.6.), w którym główna bohaterka podróży przez las. Na swojej drodze napotyka przeszkody w postaci kolców i pił które pokonuje przeskakując nad nimi. Po dotarciu do bram miasta, poziom zostaje ukończony.



Rys. 10.6. Projekt poziomu 3

## 10.7. Poziom 4 - Podróż przez miasto

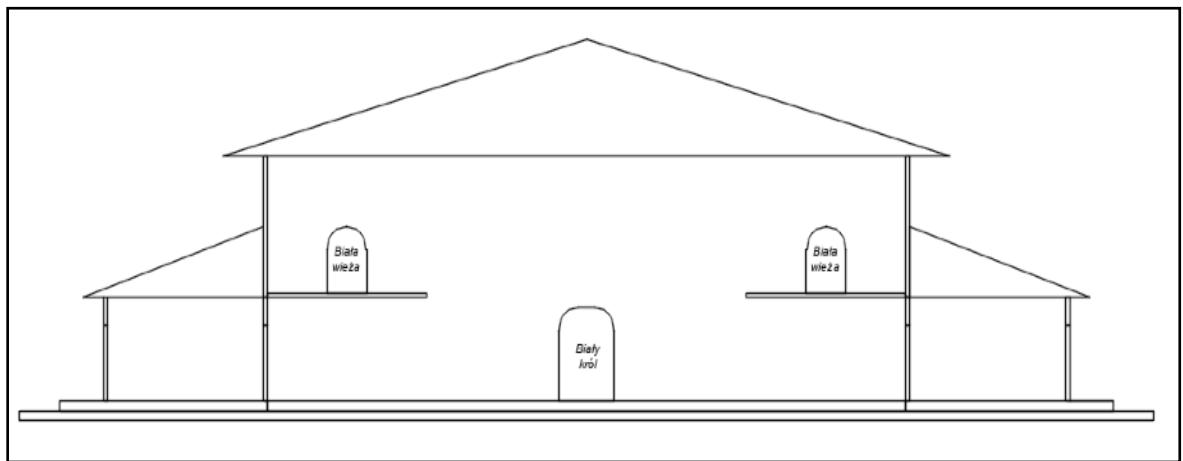
Jest to czwarty akt gry. Po wyjściu z lasu "Królowa" dociera do miasta, przez które podróżuje pokonując napotkanych przed sobą przeciwników i przeszkode. W pewnym momencie główna bohaterka dociera do katedry, gdzie spotyka kolejnego przeciwnika specjalnego: "Białego Króla Szachów". Projekt poziomu znajduje się na ilustracji poniżej (rys. 10.7.).



Rys. 10.7. Projekt poziomu 4

## 10.8. Poziom 4 - Walka w katedrze z białym królem

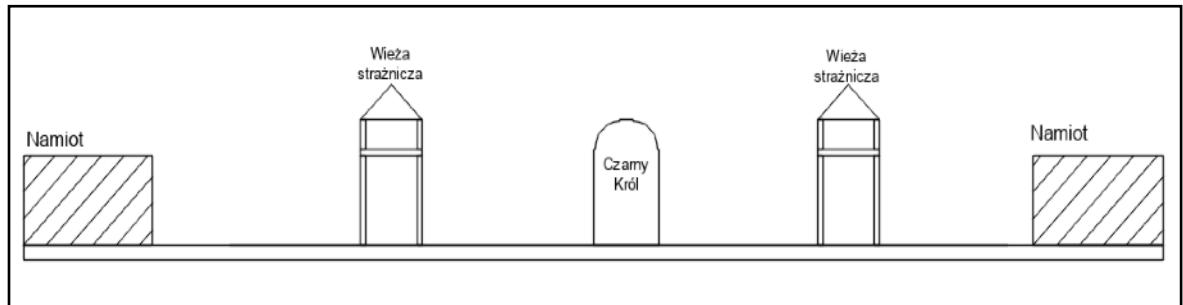
Po wejściu do katedry (projekt znajduje się na rys.10.8.), "Królowa" staje przed przeciwnikiem specjalnym - "Białym Królem Szachów". W trakcie walki "Król" przywołuje na bocznych balkonach dodatkowych przeciwników - "Białe Wieże", które strzelając do głównej bohaterki stanowią dodatkowe wyzwanie dla gracza. Po pokonaniu "Białego Króla" poziom się kończy.



Rys. 10.8. Projekt katedry

### 10.9. Poziom 5 - Walka w obozie - Walka z czarnym królem

Jest to ostatni poziom gry, gdzie "Królowa" staje przed ostatnim, najsilniejszym przeciwnikiem specjalnym - "Czarnym Królem Szachów". Walka odbywa się w obozie wojsk "Czarnego Króla" (rys.10.9.). W trakcie walki przeciwnik przywołuje Czarne Wieże na platformach wież strażniczych, które podobnie jak Białe Wieże strzelają do królowej. Poziom kończy się wraz z pokonaniem Czarnego Króla. Wraz z końcem poziomu gra się kończy.



Rys. 10.9. Projekt obozu czarnego króla

## **11. Zastosowane technologie**

### **11.1. Środowisko Unity**

Unity jest silnikiem do tworzenia gier i interaktywnych aplikacji. Został stworzony przez Unity Technologies w 2005 roku i jest rozwijany i wpierany do dzisiaj. Silnik jest dostępny na wielu platformach, między innymi na komputerach z systemami Windows, Linux oraz macOS oraz na platformach mobilnych np. iOS lub Android. Podstawowa licencja na silnik Unity jest udostępnia za darmo, jednak umowa licencyjna zawiera zapis wedle którego wymagany jest zakup płatnej licencji w przypadku gdy dany projekt oparty na silniku osiągnie przychód 200 tys. dolarów w ciągu 12 miesięcy. Postawa taka jest znaczącym wsparciem dla mniejszych zespołów i początkujących programistów dlatego że pozwala na korzystanie z profesjonalnych narzędzi bezpłatnie (stan na kwiecień 2023 roku).

Silnik Unity posiada wiele zalet do których można zaliczyć:

- Intuicyjny i prosty interfejs użytkownika
- Bogaty zestaw dołączonych narzędzi oraz dokumentacji obejmujących między innymi narzędzia do modelowania, tworzenia i edycji grafiki ,animacji oraz dźwięku czy narzędzia do zarządzania fizyką lub skryptami [7].
- Unity Asset Store [8] - jest to ogromna baza tworzona między innymi przez społeczność zawierającą modele, tekstury, skrypty a nawet gotowe projekty aplikacji. W sklepie można znaleźć zarówno płatne jak i darmowe elementy z których inni twórcy mogą korzystać i umieszczać je w swoich projektach.
- Duża społeczność rozwinięta wokół silnika tworząca różnego rodzaju poradniki, filmy czy fora na których komunikują się zarówno początkujący jak i profesjonalni programiści

### **11.2. Język C# i środowisko Microsoft Visual Studio**

Silnik Unity jako jeden z języków do tworzenia skryptów wykorzystuje język C#. Język C# jest językiem obiektowym zaprezentowanym przez Microsoft w 2000 roku. Język ten został zaprojektowany tak, aby łączyć w sobie elementy języków C++ i Java, a jednocześnie eliminować pewne ich wady. W celu poszerzenia wiedzy odnośnie języka C# wykorzystano źródła [9,10]. Obecnie na rynku jest dostępnych wiele środowisk programistycznych wspierających programowanie w języku C#, z których jednym z najpopularniejszych jest Microsoft Visual Studio. Środowisko to jest dostępne na systemach operacyjnych Windows i macOS i zawiera szereg funkcji i narzędzi z których jako najważniejszy w kontekście pisania skryptów dla silnika Unity jest integracja z silnikiem. Wśród innych funkcji środowiska znajdziemy miedzy innymi: automatyczne uzupełnianie, analiza kodu, funkcje debugowania oraz testowania, integracje z systemem kontroli wersji (m.in. Git) oraz szalony projektów.

### **11.3. Program graficzny Blender**

Blender jest programem do modelowania obiektów oraz animacji trójwymiarowych. Oprogramowanie oferuje szereg narzędzi do tworzenia modeli 3D, posiada rozbudowany interfejs. Może być to problematyczne dla nowych użytkowników, jednak nietrudno znaleźć odpowiednie poradniki [11,12], gdyż jest to jedno z najpopularniejszych darmowych oprogramowań graficznych 3D. Blender umożliwia pełną swobodę twórczą artystów, twórców gier, animatorów czy początkujących entuzjastów grafiki trójwymiarowej.

### **11.4. Platforma Mixamo**

Mixamo to platforma internetowa firmy Adobe ułatwiająca animowanie postaci humanoidalnych. Po założeniu konta użytkownik może korzystać z wszystkich funkcjonalności tej strony. Należy do nich importowanie modelów postaci oraz automatyczne generowanie dla nich szkieletu, umożliwiające dodanie animacji do postaci. Platforma udostępnia wiele podstawowych animacji tj.: bieganie, skok, ale również te bardziej zaawansowane, np taniec, animacja śmierci itp. Mixamo umożliwia dostosowywanie różnych parametrów animacji - prędkości, liczby klatek, szerokości ramion [13,14].

### **11.5. Środowisko Visual Paradigm**

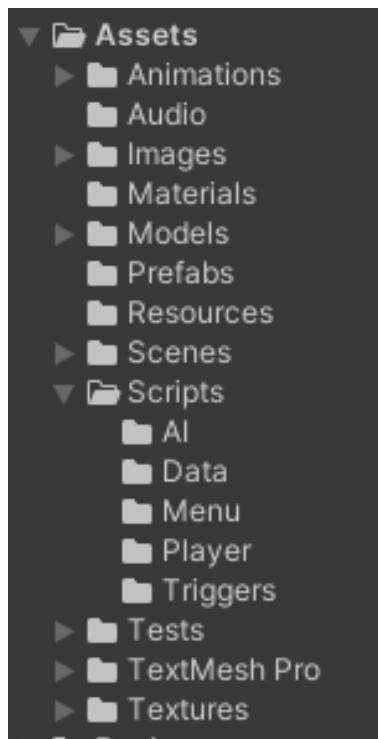
Visual Paradigm to program który umożliwia tworzenie i zarządzanie różnego rodzaju modelami biznesowymi w standardzie języka UML [15]. Narzędzie to pozwala na tworzenie diagramów takich jak np. diagram klas, lub diagram przypadków użycia.

### **11.6. System kontroli wersji Plastic SCM**

Plastic SCM to system kontroli wersji, wchodzący w skład pakietu Unity DevOps. Umożliwia zarządzanie kodem źródłowym i plikami projektu w środowisku Unity. Narzędzie to jest pomocne podczas pracy zespołowej, każdy członek zespołu może zajmować się inną funkcjonalnością w tym samym czasie, na oddzielnej gałęzi. Program umożliwia śledzenie zmian w projekcie, dzięki intuicyjnej oprawie graficznej. Plastic SCM jest narzędziem wieloplatformowym, więc użytkownicy korzystający z różnych systemów operacyjnych mogą pracować w ten sam sposób, bez żadnych problemów. W celu poszerzenia wiedzy zastosowana została dokumentacja narzędzia [16].

## 12. Organizacja pracy w środowisku UNITY

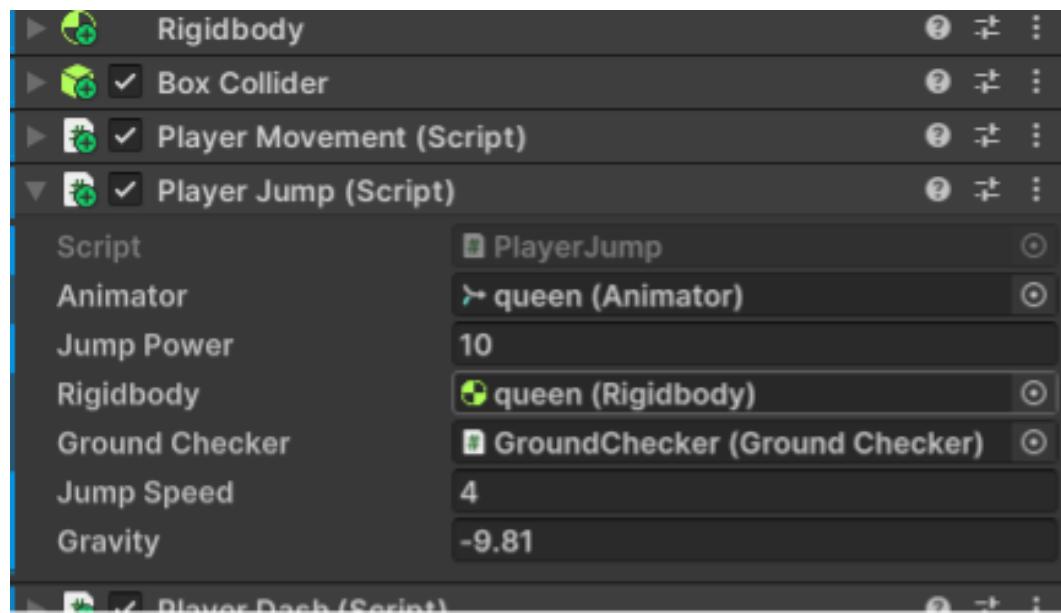
Podczas implementacji projektu, warto zachować jego uporządkowaną strukturę. Nowo utworzony projekt w Unity posiada folder „Assets” jako główny katalog, a w nim podfoldery dla różnych rodzajów zasobów (m.in.: skrypty, modele, animacje, tekstury). Taką strukturę przedstawia rysunek 12.1. Odpowiednia organizacja projektu pozwala na szybki i intuicyjny dostęp do konkretnych zasobów.



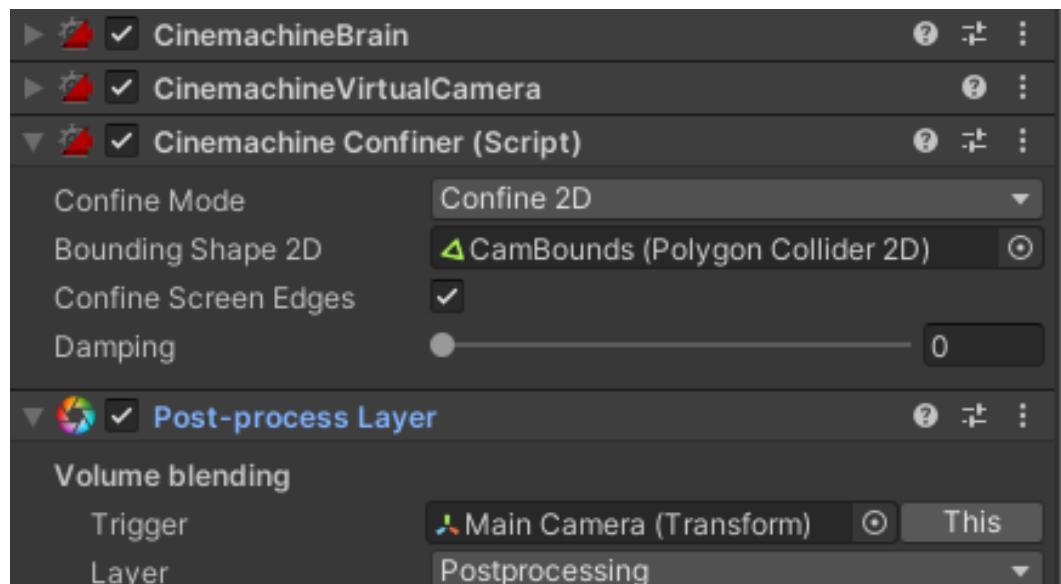
Rys. 12.1. Struktura projektu w Unity

Każdemu obiektowi można przypisać odpowiednie zależności, korzystając z wbudowanych funkcji środowiska Unity, jak i zewnętrznych rozszerzeń, które można doinstalować korzystając z narzędzia Package Manager. Rysunek 12.2. przedstawia fragment atrybutów przypisanych do postaci głównej: „Rigidbody” – odpowiadające za fizykę postaci, „Box Collider” – umożliwiający kolizje z innymi obiekttami oraz skrypty napisane w środowisku programistycznym VisualStudio. Skrypt odpowiedzialny za skok gracza, umożliwia na dostęp do pól jego klasy oraz ich modyfikacje z poziomu Unity.

Na rysunku 12.3. można zobaczyć atrybuty nałożone na kamerę główną, odpowiednio z rozszerzeń: „Cinemachine” – paczka odpowiedzialna za pracę kamery, podążanie za postacią oraz „Postprocessing” – obsługująca ustawianie jasności podczas rozgrywki. Łatwa dostępność do wielu ustawień umożliwia wygodną pracę w środowisku, gotowe do użycia narzędzie może zastąpić skrypty do tego samego przeznaczenia.

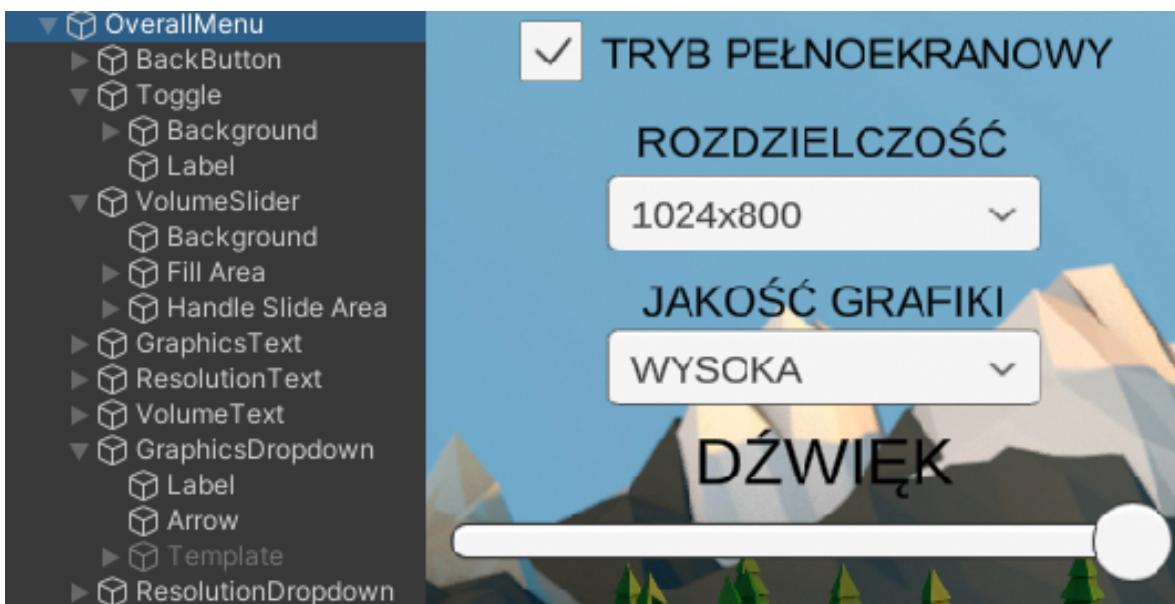


Rys.12.2. Widok atrybutów przypisanych do obiektu



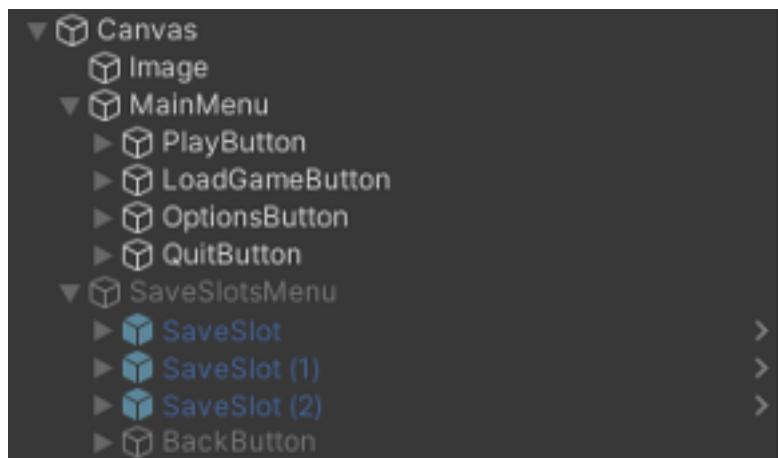
Rys.12.3. Widok atrybutów kamery

Przy tworzeniu scen menu głównego gry, ustawień, czy sceny wyboru poziomu wykorzystano zewnętrzną paczkę „TextMeshPro”, która dostarcza elementy UI takie jak: przyciski, suwaki, listy rozwijane. Rysunek 12.4. przedstawia te elementy w widoku ustawień ogólnych. Każdy z nich ma przypisaną odpowiednią funkcję ze skryptu, która jest obsługiwana przy zmianie wartości „OnValueChanged”. Rozszerzenie „TextMeshPro” umożliwia także zamianę domyślnych elementów graficznych na własne, co może urozmaicić warstwę graficzną. Elementy UI dostępne domyślnie w środowisku nie tylko mają mniej dostępnych stylów wizualnych (np. przycisk nie będzie zmieniał koloru po najechaniu na niego kursem, czy kliknięciu), ale też nie radzą sobie dobrze z dopasowywaniem do różnych rozdzielczości i skalowaniem tekstu.



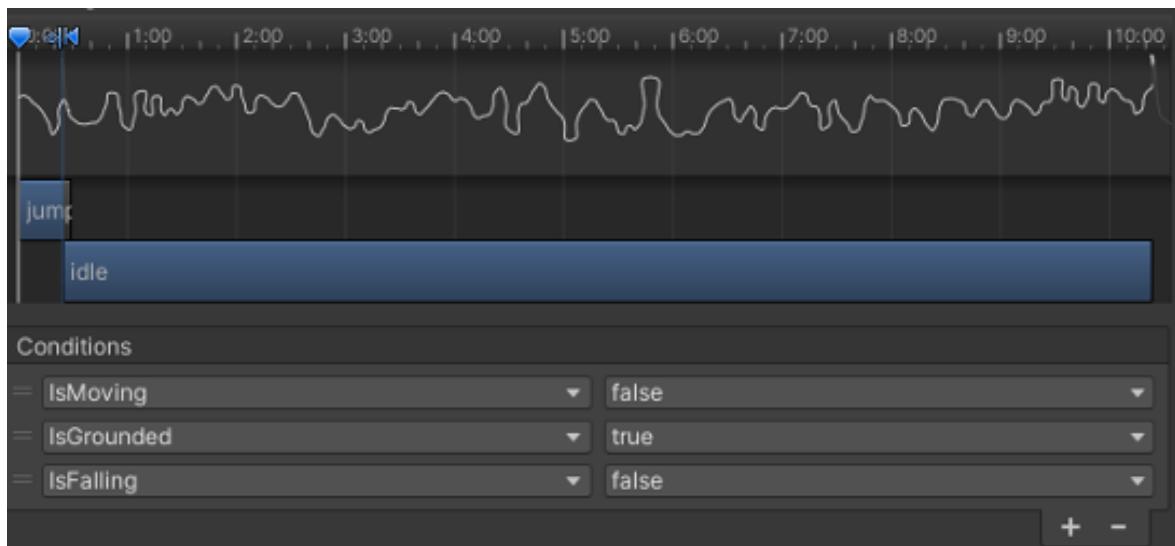
Rys. 12.4. Widok elementów menu ustawień

Obiekty na scenie mogą zostać dezaktywowane, wyświetlają się wtedy na szaro, co widać na rysunku 12.5. Takie podejście pozwala na umieszczenie kilku widoków na jednej scenie po odpowiednim oskryptowaniu obiektów. Gdy gracz zechce rozpocząć grę, obiekt „MainMenu” zostanie wyłączony, a pojawi się wybór profilu gracza „SaveSlotsMenu”.



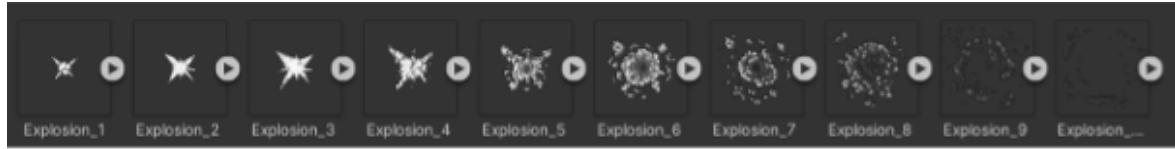
Rys. 12.5. Widok aktywnych oraz dezaktywanych elementów

Kolejnym bardzo przydatnym, wbudowanym narzędziem dostępnym w Unity jest Animator. Po utworzeniu kontrolera animacji dla danej postaci, można dodać do niej animacje oraz ustalić warunki przejścia między tymi animacjami. Rysunek 12.6. przedstawia przejście między skokiem a animacją bezczynną „idle” postaci głównej. Jeżeli postać się nie porusza oraz stoi na ziemi, animacja zostanie odpowiednio zmieniona.



Rys. 12.6. Widok ustawień animacji

Animacje mogą zostać zainportowane do projektu w formacie fbx, ale Unity także udostępnia opcję tworzenia animacji poklatkowych. Dzięki narzędziu Animation można zaznaczyć pozycję, rotację lub skalę danego obiektu w odpowiednich klatkach, aby utworzyć proste animacje. Środowisko również zaproponuje automatyczne utworzenie animacji, po przeciągnięciu kilku dwuwymiarowych obrazków jednocześnie (Rys 12.7.). Następnie gotową animację można zapisać jako wzorzec, aby mieć do niej dostęp oraz zachować ją w odpowiednim folderze.

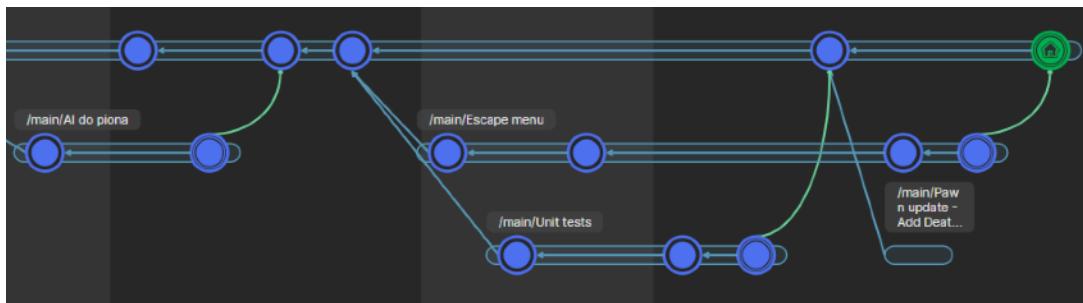


Rys. 12.7. Widok poszczególnych klatek animacji

## 13. Implementacja

### 13.1. Stworzenie projektu w systemie kontroli wersji

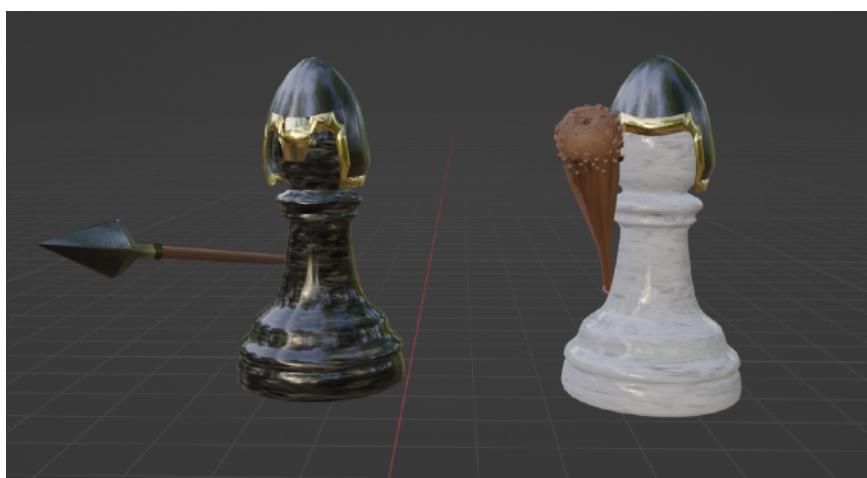
Implementację projektu rozpoczęto od utworzenia organizacji na platformie Unity Cloud, a następnie utworzenia pustego projektu aplikacji. W kolejnych krokach zostało stworzone repozytorium Unity Version Control, wykorzystujące system Plastic SCM, ułatwiający pracę grupową. Posiada on przejrzysty interfejs umożliwiający zapisywanie zmian oraz pozwala na bezpieczne ich łączenie ze zmianami innych użytkowników. Na rys. 13.1 można zaobserwować zapisane zmiany z podziałem na gałęzie oraz ich łączenie z gałęzią główną (main).



Rys. 13.1. Fragment widoku historii wersji w kliencie Plastik SCM

### 13.2. Tworzenie modeli i animacji

Modele wykorzystane na potrzeby gry, zostały stworzone w programie graficznym Blender. Blender pozwala na modelowanie, teksturowanie oraz obróbkę modeli 3D, jak i oferuje system renderowania pozwalający na generowanie obrazów i animacji. Modele stworzone w Blenderze dla oszczędności czasu i zasobów były eksportowane do platformy Mixamo w celu stworzenia dla nich animacji. Gotowy model był następnie importowany do środowiska Unity, gdzie potem mógł być już przypisany do danej sceny, oraz można było przypisać dla niego skrypty. Przykładowe modele stworzone w Blenderze zaprezentowane są na ilustracji poniżej (rys. 13.2.).



Rys. 13.2. Przykładowe modele przygotowane w programie Blender

### 13.3 Pisanie skryptów

Do implementacji skryptów Unity wykorzystuje język C#, a domyślnym środowiskiem do pisania skryptów jest środowisko programistyczne Microsoft Visual Studio. Dzięki wygodnej integracji istnieje możliwość nadania wartości z poziomu edytora Unity, korzystając z publicznych zmiennych lub atrybutów „[SerializeField]”. Ukończony i komplilujący się skrypt można przypisać do obiektów na scenie. Skrypty zawierają określone metody tj.: Start() - wywoływana podczas inicjalizacji obiektu, czy Update() - wywoływana w każdej klatce gry. Unity posiada również wiele przydatnych klas przy tworzeniu gier: „Rigidbody”, „Collider” - odpowiadające za fizykę gry, „Transform” - przemieszczanie obiektów, „SceneManagement” - zarządzanie scenami. Podczas napotkania błędów lub w celu monitorowania zachowania skryptów można skorzystać z narzędzi do debugowania.

### 13.4 Przechowywanie danych gry

Klasa „GameData” (Listning 13.1.) reprezentuje dane gry, takie jak pozycja gracza, zdrowie postaci, punkty zebrane w grze, czy liczba śmierci. Konstruktor ustawia domyślne wartości tych danych.

```
public class GameData
{
    public float health;
    public Vector3 playerPosition;
    public int pointsCollected;
    public int deathCounter;

    public GameData()
    {
        health = 6;
        playerPosition = Vector3.zero;
        pointsCollected = 0;
        deathCounter = 0;
    }
}
```

Listing 13.1. Klasa GameData

Korzystając z klasy „GameData”, w klasie „FileDataHandler” obsłużony zostaje zapis danych do pliku JSON (fragment metody Save na listningu 13.2.) oraz analogicznie odczyt tych danych z pliku. Dane są przechowywane w odpowiednim folderze w zależności od wybranego profilu gracza. Do konwersji danych na format JSON i z powrotem wykorzystano bibliotekę „JsonUtility”.

```

public void Save(GameData data, string profileId)
{
    string fullPath = Path.Combine(dataDirPath, profileId, dataFileName);
    try
    {
        Directory.CreateDirectory(Path.GetDirectoryName(fullPath));
        string dataToStore = JsonUtilityToJson(data, true);

        using (FileStream stream = new FileStream(fullPath, FileMode.Create))
        {
            using (StreamWriter writer = new StreamWriter(stream))
            {
                writer.Write(dataToStore);
            }
        }
    }
}

```

*Listing 13.2. Metoda Save (fragment)*

Klasa „DataPersistenceManager” zarządza stanami gry. Wykorzystuje klasę „FileDataHandler” do obsługi operacji zapisu i odczytu danych. Implementuje wzorzec projektowy Singleton (Listning 13.3.), aby mieć tylko jedną instancję tej klasy w grze.

```

public static DataPersistenceManager instance { get; private set; }

private void Awake()
{
    if (instance != null)
    {
        Destroy(this.gameObject);
        return;
    }

    instance = this;
    DontDestroyOnLoad(this.gameObject);

    this.dataHandler = new FileDataHandler(UnityEngine.Application.persistentDataPath, fileName);
}

```

*Listing 13.3. Klasa DataPersistenceManager*

Gra zostaje zapisywana lub wczytywana w odpowiednich momentach korzystając z metod „OnSceneLoaded”, „OnSceneUnloaded”. Zapis gry zostaje wywołany również, gdy gracz wejdzie w określony obszar ogniska (Listning 13.4.). Ponadto gracz odzyskuje wtedy wszystkie życia.

```

private void OnEnteredCampfireArea(bool entered, Vector3 position)
{
    if (entered && !usedCampfires.Contains(position))
    {

        foreach (IDataPersistence dataPersistenceObj in dataPersistenceObjects)
        {
            if (dataPersistenceObj is Character character)
            {
                character.HealToMaxHealth();
            }
        }

        SaveGame();

        usedCampfires.Add(position);
    }
}

```

*Listing 13.4. Klasa obsługująca system punktów zapisu*

## 13.5 Postać główna

Klasa „Character”, która jest nanesiona na postać główną, odpowiada za zarządzanie zdrowiem postaci (Listning 13.5.), animacjami związanymi z życiem (otrzymanie obrażeń, śmierć postaci) oraz za wczytywanie i zapisywanie stanu postaci do pliku, korzystając ze skryptu „GameData” (rozdział 13, podpunkt 3).

```

public void HealToMaxHealth()
{
    healthSystem.Health = healthSystem.MaxHealth;
    healthSystem.PrintHealth();
}

public void TakeDamage()
{
    if (!isImmune)
    {
        healthSystem.Health -= 1;
        animator.SetTrigger("DamageTrigger");
    }
}

```

*Listing 13.5. Metody klasy Charakter odpowiedzialne za zarządzanie zdrowiem postaci*

Klasą pomocniczą do zarządzania zdrowiem postaci jest „HealthSystem”. Zawiera zdarzenia „OnHealthChanged” oraz „OnDeath”, dzięki czemu można reagować na zmiany w zdrowiu w innych klasach. Posiada właściwości „Health” i „MaxHealth”, które umożliwiają dostęp do aktualnego zdrowia oraz maksymalnego zdrowia postaci. Fragment klasy przedstawiono na listingu 13.6.

```
public float Health
{
    get => health;
    set
    {
        health = value;
        if(health > maxhealth)
        {
            health = maxhealth;
        }
        if(health <= 0)
        {
            OnDeath?.Invoke();
        }

        OnHealthChanged?.Invoke();
    }
}

public float MaxHealth
{
    get => maxhealth;
    set
    {
        maxhealth = value;
    }
}
```

*Listing 13.6. Pola klasy HealthSystem*

Do postaci głównej został przypisany pusty obiekt „DamageChecker” z przypisanym skryptem o tej samej nazwie (listing 13.7.). Na obiekt został nałożony „Box Collider”, taki sam jak na postać, tylko z zaznaczoną opcją „Is Trigger”. Dzięki temu „Collider” ignoruje kolizje, a jedynie sprawdza czy coś z nim się zetknęło. Jeżeli gracz wszedł w kolizje z obiektem oznaczonym jako „Enemy” oraz zdrowie gracza jest większe od zera, wtedy gracz otrzyma obrażenia.

```

public class DamageChecker : MonoBehaviour
{
    private Character character;

    private void Start()
    {
        character = GetComponentInParent<Character>();
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Enemy") && character.GetHealth() > 0)
        {
            character.TakeDamage();
        }
    }
}

```

*Listing 13.7. Klasa DamageChecker*

```

public class GroundChecker : MonoBehaviour
{
    [SerializeField] private string groundTag = "Ground";

    private int groundedObjectsCount;

    public void OnTriggerEnter(Collider collision)
    {
        if (collision.CompareTag(groundTag))
        {
            groundedObjectsCount++;
            isGrounded = groundedObjectsCount > 0;
        }
    }

    public void OnTriggerExit(Collider collision)
    {
        if (collision.CompareTag(groundTag))
        {
            groundedObjectsCount--;
            isGrounded = groundedObjectsCount > 0;
        }
    }

    private bool isGrounded;

    public bool IsGrounded()
    {
        return isGrounded;
    }
}

```

*Listing 13.8. Klasa GroundChecker*

Na listingu powyżej (listing 13.8.) znajduje się klasa „GroundChecker”, która jest nanoszona na pusty obiekt na scenie, przypisany do postaci gracza. Obiekt ten posiada atrybut „BoxCollider”, który po odpowiednim umieszczeniu go na scenie, zwraca informacje czy gracz znajduje się na ziemi. Skrypt ten pomocny jest przy obsługiwaniu skoku gracza, ponieważ uniemożliwia skoki bez limitu. Skrypt uwzględnia, że postać może znaleźć się na więcej niż jednym obiekcie oznaczonym jako podłożo.

```
private void Update()
{
    bool isMoving = movement.IsMoving();
    animator.SetBool(_isMovingHash, isMoving);
    animator.SetBool(_isGroundedHash, groundChecker.IsGrounded());

    bool isFalling = rigidbody.velocity.y < -0.01f;
    animator.SetBool(_isFallingHash, isFalling);

    if (Input.GetKeyDown(attack) && !PauseMenu.GameIsPaused)
    {
        isAttacking = true;
        animator.SetTrigger("AttackTrigger");
        StartCoroutine(WaitForAttackAnimation());
    }

    if (isMoving && character.canMove)
    {
        float horizontalInput = movement.GetHorizontalInput();
        if (horizontalInput < 0)
        {
            transform.localScale = new Vector3(0.4f, 0.4f, -0.4f);
        }
        else if (horizontalInput > 0)
        {
            transform.localScale = new Vector3(0.4f, 0.4f, 0.4f);
        }
    }
}
```

Listing 13.9. Fragment klasy PlayerAnimation

Na listingu 13.9. przedstawiono fragment klasy „PlayerAnimation”, odpowiadającej za ustawianie prawidłowych wartości parametrów animatora (żeby każda animacja uruchamiała się w odpowiednim momencie). Metoda sprawdza w każdej klatce gry, czy gracz się nie porusza, znajduje się na ziemi, spada albo naciska przycisk myszy odpowiedzialny za atak. Skrypt „PlayerAnimation” obsługuje też obrót postaci gracza w zależności od naciśniętego klawisza (poruszania się w prawo lub lewo).

```

private void Update()
{
    float moveInput = 0f;

    if (Input.GetKey(moveLeftKey))
    {
        moveInput = -1f;
    }
    else if (Input.GetKey(moveRightKey))
    {
        moveInput = 1f;
    }

    input = new Vector2(moveInput, 0);
}

private void FixedUpdate()
{
    if (character.canMove)
    {
        Vector3 move = input * moveSpeed * Time.fixedDeltaTime;
        rigidbody.velocity = new Vector2(move.x, rigidbody.velocity.y);
    }
    else
    {
        rigidbody.velocity = Vector2.zero;
    }
}

```

*Listing 13.10. Fragment kodu odpowiedzialnego za poruszanie się*

Poruszanie gracza również opiera się na metodzie `Update()`. Jednak po wstępnych testach, sposób poruszania się okazał się nieoptymalny. Aby przemieszczanie wyglądało płynniej, skorzystano z metody `FixedUpdate()`, uzależniając poruszanie się od czasu. Dodatkowo kod widoczny na listingu 13.10. sprawdza czy gracz może się poruszać w celu uniknięcia zmiany pozycji gracza podczas animacji śmierci postaci.

Za skok gracza odpowiada klasa „`PlayerJump`”, której metody przedstawiono na listingu 13.11. Gdy postać znajduje się na ziemi może skoczyć i później dodatkowo skoczyć jeszcze raz, dzięki ustawianiu wartości zmiennej logicznej. Jeśli postać skacze, sprawdzana jest jej faza skoku (wyskok oraz opadanie), a następnie przypisana zostaje odpowiednia prędkość i siła skoku zgodnie z fizyką. Za poprawne działanie skoku odpowiada również skrypt „`GroundChecker`” (zawarty wyżej w podpunkcie tego rozdziału), którego metoda „`IsGrounded`” zwraca wartość logiczną określającą czy gracz stoi na ziemi.

```

private void Update()
{
    if (Input.GetKeyDown(jumpKey))
    {
        if (groundChecker.IsGrounded())
        {
            isJumping = true;
            canDoubleJump = true;
            animator.SetTrigger("JumpTrigger");
        }
        else if (canDoubleJump)
        {
            isJumping = true;
            canDoubleJump = false;
            animator.SetTrigger("JumpTrigger");
        }
    }
}

private void FixedUpdate()
{
    if (isJumping)
    {
        rigidbody.velocity = new Vector2(rigidbody.velocity.x, 0f);
        rigidbody.AddForce(new Vector2(0, jumpPower), ForceMode.Impulse);
        rigidbody.AddForce(new Vector2(0, gravity * jumpSpeed), ForceMode.Force);
        isJumping = false;
    }
    if (rigidbody.velocity.y < 0)
    {
        rigidbody.AddForce(new Vector2(0, gravity * jumpSpeed/2), ForceMode.Acceleration);
    }
}

```

*Listing 13.11. Metody klasy PlayerJump*

Gracz może również wykonać szarżę w wybranym kierunku, znajdując się na ziemi lub w powietrzu. Odpowiada za to skrypt „PlayerDash”, którego fragment widoczny jest na listingu 13.12. Postać wykona szarżę po podwójnym kliknięciu przypisanego do tej akcji przycisku w krótkim odstępie czasu.

```

private void Update()
{
    if (character.GetHealth() > 0)
    {
        if (Time.time - lastDashTime >= dashCooldown && Input.GetKeyDown(dash))
        {
            Dash();
        }
    }

    if (isDashing)
    {
        float dashDirection = (transform.localScale.z > 0) ? 1f : -1f;
        Vector3 dashVector = new Vector3(dashDirection, 0f, 0f) * dashDistance;
        movement.ApplyExternalForce(dashVector / dashDuration);

        isDashing = false;

        lastDashTime = Time.time;
    }
}

```

*Listing 13.12 Fragment klasy PlayerDash*

Klasa „Throwing” odpowiada za atak alternatywny gracza, którym jest wystrzelenie pocisku. Kierunek wystrzału zależy od aktualnego zwrotu postaci. Pocisk porusza się poziomo i zostaje zniszczony, gdy znajduje się poza polem widzenia gracza lub wejdzie w kolizję z dowolnym przeciwnikiem. Gracz może wystrzelić pocisk po wciśnięciu odpowiedniego klawisza (domyślnie prawy przycisk myszy) tylko wtedy, gdy postać znajduje się na ziemi i wartość logiczna flagi „readyToThrow” wynosi „true”. Po każdym wystrzale należy odczekać 10 sekund, aby móc strzelić ponownie. Listing 13.13. przedstawia warunki, przy których spełnieniu wywołana zostanie procedura „Throw” .

```

private void Update()
{
    if (Input.GetKeyDown(throwKey) && readyToThrow
        && groundChecker.IsGrounded() && !playerAnimation.isAttacking)
    {
        readyToThrow = false;

        playerAnimator.SetTrigger("ShootTrigger");

        StartCoroutine(ThrowAfterAnimation(0.5f));
        StartCoroutine(UpdateCooldownSlider());
    }
}

private IEnumerator ThrowAfterAnimation(float animationLength)
{
    yield return new WaitForSeconds(animationLength);

    Throw();
}

```

*Listing 13.13. Warunki wywołania metody Throw*

Metoda „Throw” (Listing 13.14.) zawiera kod odpowiedzialny za utworzenie pocisku na podstawie wzorca. Model postaci głównej ma przypisany pusty obiekt „shootingPoint”, którego ustawienie definiuje punkt w którym pojawi się pocisk. Jest on kulą z komponentami „Rigidbody” oraz „BoxCollider”, której zostaje przypisany wektor nadający jej odpowiednią siłę wystrzału. Zmienna „throwForce” jest publicznym polem klasy „Throwing”, co umożliwia dostosowanie prędkości pocisku z poziomu środowiska Unity w zakładce „Inspector”.

Skrypt klasy „Throwing” obsługuje również suwak widoczny na ekranie podczas rozgrywki, na którym jest odmierzany czas odnowienia pocisku. Służy do tego metoda „UpdateCooldownSlider” przedstawiona na listingu 13.15. Funkcja „Lerp” jest odpowiedzialna za przesuwanie suwaka w zależności od czasu. Parametry „-10f” oraz „0f” są skrajnymi wartościami suwaka. Gdy wartość minimalna jest mniejsza od zera, domyślna pozycja suwaka znajdzie się po jego prawej stronie. Po wystrzale pocisku, suwak zostaje zresetowany w metodzie „ResetThrow”.

```

private void Throw()
{
    GameObject projectile = Instantiate(objectToThrow, attackPoint.position, attackPoint.rotation);
    Rigidbody projectileRb = projectile.GetComponent< Rigidbody >();

    Vector3 forceDirection;

    if (playerTransform.localScale.z > 0f)
    {
        forceDirection = attackPoint.transform.forward;
    }
    else
    {
        forceDirection = -attackPoint.transform.forward;
    }

    Vector3 forceToAdd = forceDirection * throwForce + attackPoint.transform.up * throwUpwardForce;
    projectileRb.AddForce(forceToAdd, ForceMode.Impulse);

    StartCoroutine(ResetThrow());
}

```

*Listing 13.14. Metoda Throw*

```

private IEnumerator ResetThrow()
{
    yield return new WaitForSeconds(throwCooldown);

    readyToThrow = true;

    if (cooldownSlider != null)
    {
        cooldownSlider.value = 0f;
    }
}

private IEnumerator UpdateCooldownSlider()
{
    float elapsedTime = 0f;

    while (elapsedTime < throwCooldown)
    {
        if (cooldownSlider != null)
        {
            cooldownSlider.value = Mathf.Lerp(-1f, 0f, elapsedTime / throwCooldown);
        }

        elapsedTime += Time.deltaTime;
        yield return null;
    }

    if (cooldownSlider != null)
    {
        cooldownSlider.interactable = false;
    }
}

```

*Listing 13.15. Fragment klasy Throw odpowiedzialny za obsługę suwaka*

## 13.6 Menu i ustawienia

Podczas rozgrywki, po naciśnięciu klawisza „ESC”, wyświetla się menu pauzy. Odpowiada za to skrypt „PauseMenu”. W menu pauzy gracz może wznowić grę, wejść w ustawienia lub wyjść do menu głównego. Na listingu 13.16. przedstawiono metody odpowiedzialne za wstrzymanie i wznowienie rozgrywki.

```

void Update()
{
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        if (GameIsPaused)
        {
            Resume();
        }
        else
        {
            Pause();
        }
    }
}

public void Resume()
{
    pauseMenuUI.SetActive(false);
    options.SetActive(false);
    Time.timeScale = 1f;
    GameIsPaused = false;
}

void Pause()
{
    pauseMenuUI.SetActive(true);
    Time.timeScale = 0f;
    GameIsPaused = true;
}

```

*Listing 13.16. Metody skryptu PauseMenu*

W ustawieniach menu pauzy gracz może m.in. dostosować jasność, dzięki skryptowi „Brightness” dołączonemu do suwaka (fragment skryptu na listingu 13.17.) oraz zewnętrznej bibliotece „Postprocessing” dostępnej w środowisku Unity.

```

void Start()
{
    brightness.TryGetSettings(out exposure);
    AdjustBrightness(brightnessSlider.value);
}

public void AdjustBrightness(float value)
{
    if (value != 0)
    {
        exposure.keyValue.value = value;
    }
    else
    {
        exposure.keyValue.value = .05f;
    }
}

```

*Listing 13.17. Skrypt Brightness*

Po uruchomieniu gry wyświetla się menu główne, w którym gracz może rozpoczęć nową grę, wczytać rozgrywkę, przejść do ustawień lub wyjść z gry. Odpowiadają za to poszczególne metody klasy „Menu”, przedstawione na listingu 13.18.

```
public void PlayGame()
{
    for (int i = 1; i <= 4; i++)
    {
        PlayerPrefs.SetInt("ResetPointsFlag_Level_" + i, 1);
    }
    PlayerPrefs.SetInt("NewGameClicked", 1);

    saveSlotMenu.ActivateMenu(false);
    this.DeactivateMenu();
}

public void OnLoadGame()
{
    saveSlotMenu.ActivateMenu(true);
    PlayerPrefs.DeleteKey("NewGameClicked");
    for (int i = 1; i <= 4; i++)
    {
        PlayerPrefs.DeleteKey("ResetPointsFlag_Level_" + i);
    }
    this.DeactivateMenu();
}

public void GoToSettingsMenu()
{
    SceneManager.LoadScene("SettingsMenu");
}

public void QuitGame()
{
    Application.Quit();
}
```

Listing 13.18. Metody klasy Menu

```
public void SetResolution(int resolutionIndex)
{
    Resolution resolution = resolutions[resolutionIndex];
    Screen.SetResolution(resolution.width, resolution.height, Screen.fullScreen);
}

public void SetQuality(int qualityIndex)
{
    QualitySettings.SetQualityLevel(qualityIndex);
}

public void SetFullscreen(bool isFullscreen)
{
    Screen.fullScreen = isFullscreen;
}

public void SetVolume(float volume)
{
    audioMixer.SetFloat("volume", volume);
}
```

Listing 13.19. Metody obsługujące ustawiania aplikacji

W ustawieniach gracz może włączyć lub wyłączyć tryb pełnoekranowy, dostosować rozdzielcość, głośność dźwięku oraz jakość grafiki. Odpowiednie metody przedstawione na listingu 13.19. są przypisane do odpowiadających im elementów UI (tj.: suwak, lista rozwijana).

W skrypcie „SaveSlot” (listing 13.20.) metoda „Awake” pobiera referencję do przycisku, na którym znajduje się ten skrypt. Korzystając z zapisanych danych, zostają wyświetlane odpowiednie informacje o postępie w rozgrywce. Metoda „SetInteractable” umożliwia wyłączenie interakcji z przyciskiem, gdy zajdzie taka potrzeba (pusty profil przy wczytaniu gry).

```
private Button saveSlotButton;

private void Awake()
{
    saveSlotButton = this.GetComponent<Button>();
}

public void SetData(GameData data)
{
    if (data == null)
    {
        noDataContract.SetActive(true);
        hasDataContract.SetActive(false);
    }
    else
    {
        noDataContract.SetActive(false);
        hasDataContract.SetActive(true);

        percentageCompleteText.text = "Ukończono: " + data.GetPercentageComplete() + "%";
        currentHealthText.text = "Liczba życia: " + data.health;
    }
}

public string GetProfileId()
{
    return this.profileId;
}

public void SetInteractable(bool interactable)
{
    saveSlotButton.interactable = interactable;
}
```

Listing 13.20. Fragment skryptu SaveSlot

Skrypt „SaveSlotMenu” zawiera m.in. przekierowanie do sceny wyboru poziomu po wybraniu profilu gracza, powrotu do menu głównego, czy dezaktywowania przycisków w menu podczas lądowania. Metoda „OnSaveSlotClicked” zaprezentowana na listingu 13.21 jest przypisana do przycisków wyboru profilu gracza. Za pomocą wartości przechowywanej w „PlayerPrefs” (rozdział 6, podpunkt 2), procedura sprawdza czy gracz nacisnął przycisk „Nowa gra”. W takim przypadku dane o zebranych punktach oraz odblokowanych poziomach zostaną zresetowane.

```

public void OnSaveSlotClicked(SaveSlot saveSlot)
{
    string profileId = saveSlot.GetProfileId();

    DisableMenuButtons();

    DataPersistenceManager.instance.ChangeSelectedProfileId(saveSlot.GetProfileId());

    if (PlayerPrefs.GetInt("NewGameClicked", 0) == 1)
    {
        PlayerPrefs.DeleteKey("UnlockedLevel");
        PlayerPrefs.DeleteKey("NewGameClicked");

        for (int levelId = 1; levelId <= 4; levelId++)
        {
            string levelTag = "Level" + levelId;
            PlayerPrefs.DeleteKey(profileId + "_" + levelTag + "_StarsCollected");
        }
    }

    if (!isLoadingGame)
    {
        DataPersistenceManager.instance.NewGame();
    }

    SceneManager.LoadSceneAsync("LevelsPanel");
}

```

*Listing 13.21. Metoda OnSaveSlotClicked*

Skrypt „LevelMenu” obsługuje scenę wyboru poziomu. Przy każdym załadowaniu sceny (po wyborze profilu gracza lub ukończeniu poziomu) zostanie wywołana metoda „Awake” (listing 13.22). Pobiera ona informacje o odblokowanych poziomach z „PlayerPrefs”, na podstawie których umożliwia interakcje z przyciskami przekierowującymi na dane poziomy.

```

private void Awake()
{
    int unlockedLevel = PlayerPrefs.GetInt("UnlockedLevel", 1);
    for (int i = 0; i < buttons.Length; i++)
    {
        buttons[i].interactable = false;
    }
    for (int i = 0; i < unlockedLevel; i++)
    {
        buttons[i].interactable = true;
    }
}

```

*Listing 13.22. Kod skryptu LevelMenu*

Każdy z poziomów posiada odpowiadający do niego przycisk na scenie. Pod każdym z tych przycisków wyświetlane są gwiazdki, które ilustrują liczbę zebranych punktów przez gracza na danym poziomie. Funkcjonalność ta została obsłużona przez metodę „DisplayStars” (listning 13.23.). Obraz gwiazdkowy jest umieszczany w odpowiednim miejscu, dzięki przypisaniu go do pola tekstowego „starsCountText”.

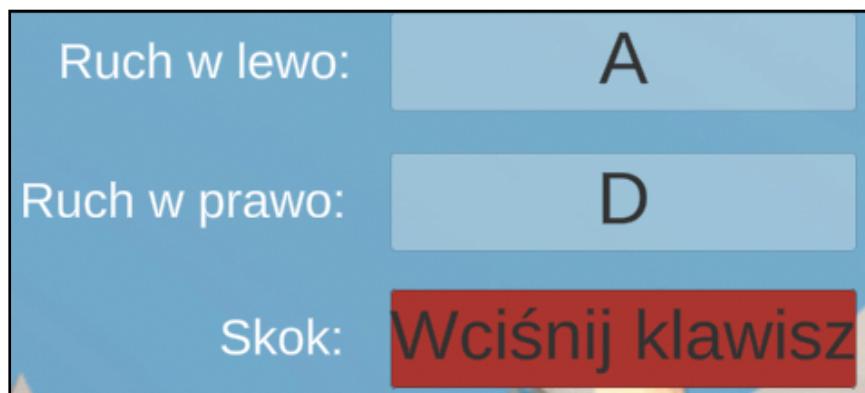
```
private void DisplayStars(TextMeshProUGUI starsCountText, int points)
{
    foreach (Transform child in starsCountText.transform)
    {
        Destroy(child.gameObject);
    }

    for (int i = 0; i < points; i++)
    {
        Image starImage = Instantiate(starImagePrefab, starsCountText.transform);

        float xPos = i * (starImage.rectTransform.sizeDelta.x + starSpacing);
        starImage.rectTransform.anchoredPosition = new Vector2(xPos, 0f);
    }
}
```

*Listing 13.23. Metoda DisplayStars*

W ustawieniach gry została zaimplementowana opcja przypisania klawiszy do danych akcji (Rys. 13.3.). Do pobierania informacji o zmianie ustawień zastosowano przyciski z zewnętrznej biblioteki „TextMeshPro” (rozdział 12).



*Rys. 13.3. Widok przypisania klawiszy w ustawieniach sterowania*

Gra posiada domyślne ustawienia sterowania, które gracz może dostosować według własnych preferencji (listning 13.24. przedstawia automatyczne ustawienie opcji sterowania). Po zmianie ustawień sterowania, zostaną one zapisane i będą zapamiętane przy kolejnym uruchomieniu gry. Jeżeli dana akcja nie będzie miała przypisanego kodu klawisza, to automatycznie zostanie przypisany domyślny klawisz do tej akcji.

Po naciśnięciu wybranej akcji, na przycisku pojawi się napis „Wciśnij klawisz”. Wtedy skrypt „Keybind” (fragment widoczny na listngu 13.25.) oczekuje na wcisnięcie przycisku myszy lub klawiatury przez gracza, aby go odpowiednio przypisać oraz wyświetlić kod wybranego klawisza. Skrypt ten jest przypisany do pustych elementów na scenie, które pobierają kod klawisza do danej akcji z tekstu wyświetlonego na przycisku.

```

void Start()
{
    playerPrefsKey = "CustomKey_" + gameObject.name;

    if (buttonLbl != null)
    {
        switch (gameObject.name)
        {
            case "MoveLeft":
                defaultKeyCode = KeyCode.A;
                break;
            case "MoveRight":
                defaultKeyCode = KeyCode.D;
                break;
            case "Jump":
                defaultKeyCode = KeyCode.Space;
                break;
            case "Attack":
                defaultKeyCode = KeyCode.Mouse0;
                break;
            case "Dash":
                defaultKeyCode = KeyCode.LeftShift;
                break;
            default:
                defaultKeyCode = KeyCode.Space;
                break;
        }
    }

    if (PlayerPrefs.HasKey(playerPrefsKey))
    {
        buttonLbl.text = PlayerPrefs.GetString(playerPrefsKey);
    }
    else
    {
        buttonLbl.text = defaultKeyCode.ToString();
        PlayerPrefs.SetString(playerPrefsKey, defaultKeyCode.ToString());
        PlayerPrefs.Save();
    }
}

```

*Listing 13.24. Skrypt przypisujący domyślne ustawienia sterowania*

```

void Update()
{
    foreach (KeyCode keycode in Enum.GetValues(typeof(KeyCode)))
    {
        if (Input.GetKey(keycode) && buttonLbl.text == "Wciśnij klawisz")
        {
            buttonLbl.text = keycode.ToString();
            PlayerPrefs.SetString(playerPrefsKey, keycode.ToString());
            PlayerPrefs.Save();
        }
    }
}

public void ChangeKey()
{
    buttonLbl.text = "Wciśnij klawisz";
}

public KeyCode GetKey()
{
    return (KeyCode)Enum.Parse(typeof(KeyCode), buttonLbl.text);
}

```

*Listing 13.25. Fragment skryptu przespania klawisza do ustawienia*

Na ekranie rozgrywki wyświetlają się m.in. grafiki serc, przedstawiające obecną liczbę życia gracza. Do projektu dodano trzy grafiki – pełnego, połowy oraz pustego serca. Za dostęp do odpowiedniego obrazu odpowiada klasa „HealthHeart” (listing 13.26.). Zastosowano typ wyliczeniowy, który przechowuje informacje o obecnym statusie poziomu zdrowia.

```
public class HealthHeart : MonoBehaviour
{
    public Sprite fullHeart, halfHeart, emptyHeart;
    Image heartImage;

    private void Awake()
    {
        heartImage = GetComponent<Image>();
    }

    public void SetHeartImage(HeartStatus status)
    {
        switch (status)
        {
            case HeartStatus.Empty:
                heartImage.sprite = emptyHeart;
                break;
            case HeartStatus.Half:
                heartImage.sprite = halfHeart;
                break;
            case HeartStatus.Full:
                heartImage.sprite = fullHeart;
                break;
        }
    }

    public enum HeartStatus
    {
        Empty = 0,
        Half = 1,
        Full = 2
    }
}
```

Listing 13.26. Skrypt HealthHeart

Za wyświetlanie aktualnej liczby życia odpowiada metoda „DrawHearts” klasy „HealthBar”, korzystająca z przedstawionej powyżej klasy „HealthHeart”. Procedura zostaje wywoływana w momentach zmiany liczby życia gracza, tj.: otrzymanie obrażeń, odrodzenie postaci. Liczba utworzonych grafik jest zależna od maksymalnej wartości życia gracza (np. dla wartości „maxHealth” równej sześć, zostaną utworzone trzy obrazki serc,

ponieważ każde serce jest reprezentacją dwóch punktów zdrowia). Na listingu 13.27. przedstawiono opisaną metodę oraz momenty jej wywołania – podczas rozpoczęcia gry lub zmiany wartości zdrowia.

```
private void OnEnable()
{
    Character.OnPlayerDamaged += DrawHearts;
    Character.OnPlayerRespawned += DrawHearts;
}

private void OnDisable()
{
    Character.OnPlayerDamaged -= DrawHearts;
    Character.OnPlayerRespawned -= DrawHearts;
}

private void Start()
{
    DrawHearts();
}

public void DrawHearts()
{
    ClearHearts();

    float maxHealthRemainder = character.maxHealth % 2;
    float heartsToMake = (character.maxHealth / 2 + maxHealthRemainder);
    for (int i = 0; i < heartsToMake; i++)
    {
        CreateEmptyHeart();
    }

    for (int i = 0; i < hearts.Count; i++)
    {
        int heartStatusRemainder = (int)Mathf.Clamp(character.HealthSystem.health - (i*2), 0, 2);
        hearts[i].SetHeartImage((HeartStatus)heartStatusRemainder);
    }
}
```

Listing 13.27. Metoda DrawHearts

Poziomy w grze składają się z dwóch scen: plansza z podstawowymi przeciwnikami oraz punktami (rozdział 13, podpunkt 7) i arena z przeciwnikiem specjalnym. Zaimplementowano skrypt „ArenaEntrance” (listing 13.28.), żeby zrealizować przejście między scenami podczas rozgrywki. Skrypt został nanesiony na pusty obiekt na scenie, posiadający atrybut „BoxCollider” z opcją „IsTrigger”, dzięki czemu gracz zostanie załadowany na nową scenę po wejściu z nim w kolizję.

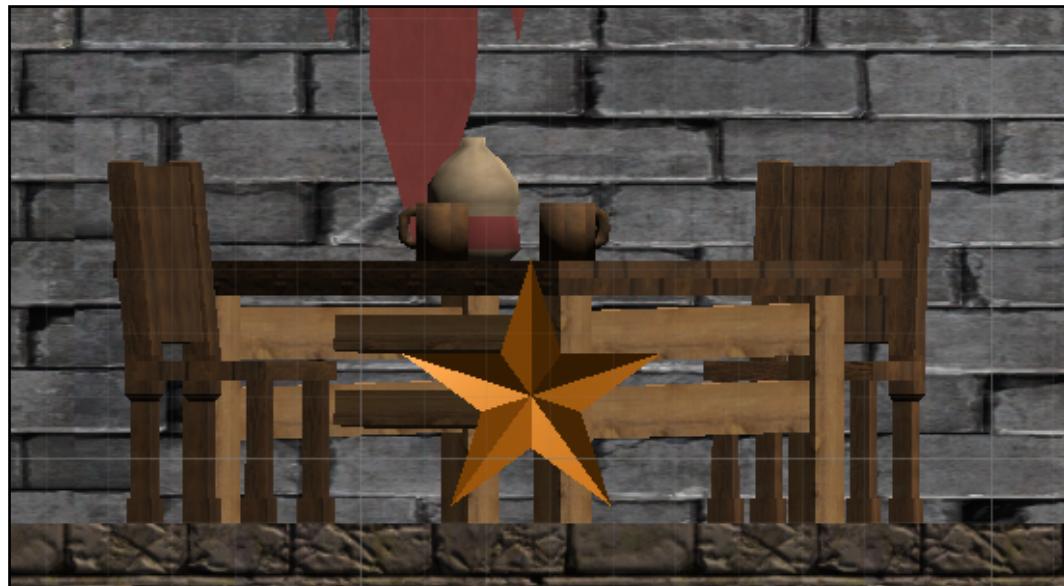
```
public class ArenaEntrance : MonoBehaviour
{
    public string bossArenaSceneName = "Boss-Fight";

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            SceneManager.LoadScene(bossArenaSceneName);
        }
    }
}
```

Listing 13.28. Klasa ArenaEntrance

## 13.7 System punktacji

Głównym skryptem odpowiedzialnym za system punktacji w grze jest klasa „Point”. Na scenach poziomów zostały rozmieszczone modele gwiazd (Rys. 13.4.), które symbolizują punkty. Jest to obiekt z komponentem „BoxCollider” o wartości „IsTrigger” ustawionej na „true” (w celu sprawdzenia, czy gracz wszedł z tym obiektem w kolizję). Ponadto gwiazda posiada prostą animację, dzięki której obraca się wokół własnej osi. Na podstawie danego modelu utworzono wzorzec, aby ułatwić rozmieszczenie pozostałych punktów na scenach.



Rys. 13.4. Przykładowy model gwiazdy

Punkty są umieszczone na różnych poziomach, dlatego po wczytaniu rozgrywki wywołana zostaje metoda „Awake” (listing 13.29.), która po krótkim opóźnieniu (w celu poprawnego załadowania elementów sceny) pobiera nazwę oznaczenia punktu. Oznaczenie obiektu, zależy od poziomu, na którym on się znajduje. Takie podejście pozwala na poprawne wyświetlanie zebranych punktów na danym poziomie (rozdział 16, podpunkt 3). Jeżeli określona gwiazdka została zebrana przez gracza, nie pojawi się ona przy kolejnym wczytaniu gry. Po rozpoczęciu nowej rozgrywki punkty zostaną zresetowane, dzięki czemu gwiazdki znów będą widoczne.

Gdy gracz wejdzie w obszar „BoxCollidera” gwiazdki, wywołana zostanie procedura „CollectPoint”. Wartość flagi przechowującej informację o zebraniu punktu zostanie ustawiana na „true”, a następnie zostanie wywołana metoda „IncreasePoints” klasy „PointCounter”. Ogólna liczba zebranych punktów zostaje zapisana w plikach zawierających dane gry, dzięki skryptom za to odpowiedzialnym (rozdział 13, podpunkt 4.). Zebrana gwiazdka zostaje dezaktywowana, a w jej pozycji zostaje uruchomiony efekt zebrania punktu, analogiczny do efektu zakończenia poziomu, który został opisany pod koniec rozdziału 15, podpunktu 3.1. (efekty różnią się kolorem).

```

void Awake()
{
    StartCoroutine(FindDataPersistenceManager());
}

private IEnumerator FindDataPersistenceManager()
{
    yield return new WaitForSeconds(0.1f);
    dataPersistenceManager = FindObjectOfType<DataPersistenceManager>();
    profileId = DataPersistenceManager.instance.GetSelectedProfileId();
    levelTag = gameObject.tag;
    currentLevel = int.Parse(levelTag.Substring(5));

    if (PlayerPrefs.GetInt("ResetPointsFlag_Level_" + currentLevel, 0) == 1)
    {
        ResetPoints();
        PlayerPrefs.SetInt("ResetPointsFlag_Level_" + currentLevel, 0);
        PlayerPrefs.Save();
    }
    else
    {
        isCollected = PlayerPrefs.GetInt(PlayerPrefsKey, 0) == 1;

        if (isCollected)
        {
            gameObject.SetActive(false);
        }
    }
}

```

*Listing 13.29. Metoda odpowiedzialna za wyświetlanie gwiazd na mapie*

```

void OnTriggerEnter(Collider col)
{
    if (col.gameObject.CompareTag("Player"))
    {
        CollectPoint();
    }
}

void CollectPoint()
{
    isCollected = true;
    PlayerPrefs.SetInt(PlayerPrefsKey, 1);
    PlayerPrefs.Save();

    PointCounter.instance.IncreasePoints(value, profileId, levelTag);

    SaveCollectedPoints();

    if (collectParticle != null && particleSpawnPoint != null)
    {
        StartCoroutine(PlayCollectParticle());
    }

    gameObject.SetActive(false);
}

```

*Listing 13.30. Metoda odpowiedzialna za zbieranie gwiazd*

Metoda „ResetPoints” (listing 13.31.) jest wywoływaną, jeżeli gracz rozpoczął nową grę na profilu, który przechowywał dane z ostatniej rozgrywki. Ustawia on wartość flagi „isCollected” na „false” oraz aktywuje widoczność gwiazdek na scenie.

```
public static void ResetPoints()
{
    Point[] points = FindObjectsOfType<Point>();

    foreach (Point point in points)
    {
        point.isCollected = false;
        PlayerPrefs.SetInt(point.PlayerPrefsKey, 0);
        PlayerPrefs.Save();

        point.gameObject.SetActive(true);
    }
}
```

Listing 13.31. Metoda ResetPoints

Klasa „PointCounter” odpowiada za aktualizację wartości zebranych punktów. Metoda „IncreasePoints” (listing 13.32.) sumuje liczbę zebranych punktów na danym poziomie. Służy do tego słownik „levelPoints”, gdzie kluczem jest łańcuch znaków „string”. Dzięki temu informacja o liczbie zebranych punktów przechowywana jest w „PlayerPrefs” i może być zastosowana do wyświetlania gwiazdek na scenie wyboru poziomu (rozdział 16, podpunkt 3.).

```
public void IncreasePoints(int p, string profileId, string levelTag)
{
    string key = profileId + "_" + levelTag + "_StarsCollected";

    if (!levelPoints.ContainsKey(key))
    {
        levelPoints[key] = 0;
    }

    levelPoints[key] += p;
    UnityEngine.Debug.Log(levelPoints[key]);
    currentPoints += p;
    PlayerPrefs.SetInt(key, PlayerPrefs.GetInt(key, 0) + p);
    PlayerPrefs.Save();
}
```

Listing 13.32. Metoda Increase Points

Zmienna „currentPoints” przechowuje ogólną wartość zebranych punktów na wszystkich poziomach. Wartość ta jest zapisywana oraz pobierana z plików gry, dzięki czemu może być wyświetlana na ekranie wyboru profilu gracza (rozdział 16, podpunkt 2.). Listing 13.33. przedstawia metody „LoadData” oraz „SaveData”, które są odpowiednio wywoływanie przy wczytaniu i zapisie gry. Jeżeli gracz rozpocznie nową grę, wartość zebranych punktów zostanie zresetowana przez ustawienie jej na „0”.

```
public void LoadData(GameData data)
{
    currentPoints = data.pointsCollected;
}

public void SaveData(ref GameData data)
{
    data.pointsCollected = currentPoints;
}

public void ResetPoints()
{
    levelPoints.Clear();
    currentPoints = 0;
}
```

*Listing 13.33 Metody odpowiedzialna za za obsługę punktów*

# 14 Ogólne ustawienia przeciwników

## 14.1. Otrzymywanie obrażeń przez przeciwników

Wywołanie metody zadającej obrażenia przeciwnikom jest metodą uniwersalną i każdy ze skryptów przeciwników uwzględnia metodę „TakeDamage”, która zostaje wywołana w ten sam sposób. Metoda jako parametr przyjmuje zmienną typu „Collider”, dzięki której rozróżnia obiekt z którym przeciwnik wszedł w kolizję. Ponadto sprawdza czy jest wykonywana animacja ataku (listing 14.1.).

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Weapon") && playerAnimation.isAttacking)
    {
        TakeDamage(1);
        playerAnimation.isAttacking = false;
    }
    else if (other.CompareTag("Projectile"))
    {
        TakeDamage(3);
    }
}
```

Listing 14.1. Wywołanie metody TakeDamage

W zależności od typu przeciwnika metoda zadająca obrażenia „TakeDamage” może się różnić np. „Leniwy Naczelnik Więzienia” posiada rozbudowaną metodę „TakeDamage” (rozdział 15, podpunkt 3.1), jednak w przypadku przeciwników podstawowych jest to prosta metoda zaprezentowana na listingu 14.2.

```
public void TakeDamage(int damage)
{
    hp -= damage;

    if (hp <= 0)
    {
        Invoke(nameof(DestroyEnemy), 0.5f);
    }
}
```

Listing 14.2. Metoda TakeDamage

## 14.2. Niszczenie obiektu przeciwnika

Niszczenie obiektu przeciwnika zostaje wywołane w momencie gdy wartość jego zdrowia będzie mniejsza bądź równa zeru, wtedy też metoda „DestroyEnemy” zatrzymuje wykonywanie wszelkich działań przez obiekt za pomocą ustawienia flagi „isAlive” na „false”, następnie zostaje wywołana animacja śmierci (Listing 14.3.).

```
private void DestroyEnemy()
{
    UnityEngine.Debug.Log("DestroyEnemy method executed");
    isAlive = false;
    animator.SetTrigger("pawnDeathTrigger");

    Vector3 enemyPosition = transform.position;

    float yOffset = 2.3f;
    enemyPosition.y += yOffset;

    StartCoroutine(WaitForDeathAnimation(enemyPosition));
}
```

*Listing 14.3. Metoda DestroyEnemy*

W celu opóźnienia zniszczenia obiektu wykorzystano interfejs IEnumarator „WaitForDeathAnimation” (listing 14.4.). Opóźnienie zostaje wywołane na czas trwania animacji śmierci. Czas trwania animacji zostaje pobrany za pomocą narzędzia animator.

```
private IEnumerator WaitForDeathAnimation(Vector3 enemyPosition)
{
    yield return new WaitForSeconds(animator.GetCurrentAnimatorStateInfo(0).length);

    Instantiate(explosionEffectPrefab, enemyPosition, Quaternion.identity);
    yield return new WaitForSeconds(0.2f);
    Destroy(gameObject);
}
```

*Listing 14.4. Interfejs WaitForDeathAnimation*

Po wywołaniu animacji śmierci przez animatora w miejscu gdzie został pokonany przeciwnik zostaje wywołana animacja wzorca wraz z poprawionymi współrzędnymi. Następnie zostaje dodane ponownie krótkie opóźnienie po którego upłynięciu obiekt zostaje zniszczony. Efekt wywołania animacji jest widoczny na rys. 14.1.

Zadawanie obrażeń graczowi zostało opisane w części dotyczącej postaci (rozdział 13, podpunkt 5). Działa ono analogicznie do sposobu przedstawionego do zadawania obrażeń przeciwnikom. Jednak w tym przypadku jest porównywany „Collider” gracza z oznaczeniem „Enemy”.



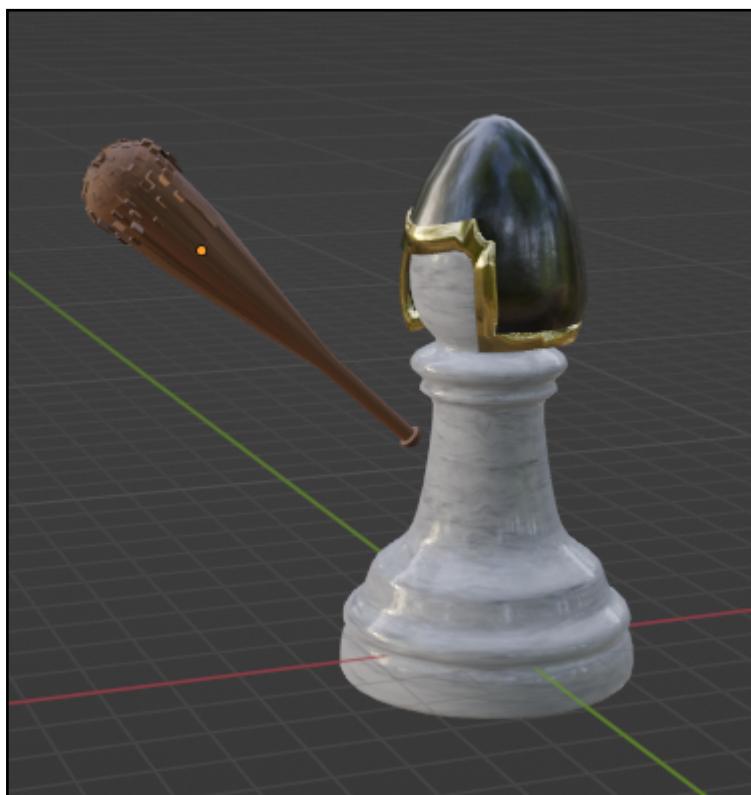
Rys. 14.1. Wywołanie animacji śmierci oraz wzorca na scenie testowej

## 15 Implementacja przeciwników

### 15.1. Implementacja przeciwników podstawowych

#### 15.1.1. Biały Pion

Na potrzeby utworzenia modelu „Białego Piona” oraz jego broni zaprezentowanych na Rys. 15.1. został wykorzystany program do modelowania grafik 3D Blender. Utworzenie animacji zostało wykonane w Unity z zastosowaniem narzędzia animator.



Rys. 15.1. Model białego piona

**Przemieszczanie się:**

**Poruszanie między punktami**

W celu umożliwienia przemieszczania się między punktami powstał kod zawarty na listingu 15.1. Metoda „MovingBetweenPoint” sprawdza za pomocą metody „PlayerInSight” czy gracz znajduje się w zasięgu ataku. Jeżeli zostanie zwrócona wartość „false” wtedy też Pion porusza się pomiędzy określonymi punktami. Gdy Pion jest w trakcie patrolowania, w momencie osiągnięcia punktu docelowego, zmieniana jest wartość składowej x wektora prędkości na ujemną w celu przemieszczenia obiektu w przeciwnym kierunku. Wymusza to umieszczenie punktu B jako punktu znajdującego się po prawej stronie od punktu A. Wówczas następuje sprawdzenie czy odległość między obiektem, a aktualnym punktem docelowym jest zbliżona, jeżeli tak wtedy punkt docelowy zostaje zmieniony na przeciwny.

```

private void MovingBetweenPoints()
{
    if (!PlayerInSight())
    {
        FaceCurrentPoint();
        Vector3 point = currentPoint.position - transform.position;

        if (currentPoint == pointB.transform)
        {
            rb.velocity = new Vector3(speed, 0f, 0f);
        }
        else
        {
            rb.velocity = new Vector3(-speed, 0f, 0f);
        }

        if (Vector3.Distance(transform.position, currentPoint.position) < 1f)
        {
            if (currentPoint == pointB.transform)
            {
                currentPoint = pointA.transform;
            }
            else
            {
                currentPoint = pointB.transform;
            }
        }
    }
}

```

*Listing 15.1. Metoda MovingBetweenPoints*

Kod ten odpowiada jednak jedynie za poruszanie się, a więc model danego obiektu był zwrócony w jedną stronę. W celu rozwiązania tego problemu powstała metoda „FaceCurrentPoint” listing 15.2. która jest wywoływaną podczas patrolowania.

```

private void FaceCurrentPoint()
{
    if (currentPoint != null)
    {
        Vector3 directionToFace = currentPoint.position - transform.position;
        directionToFace.y = 0;
        directionToFace = Quaternion.Euler(0, 90, 0) * directionToFace;
        if (directionToFace != Vector3.zero)
        {
            Quaternion targetRotation = Quaternion.LookRotation(directionToFace);
            transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation, Time.deltaTime * rotationSpeed);
        }
    }
}

```

*Listing 15.2. Metoda FaceCurrentPoint*

W celu zapobiegania błędów metoda najpierw sprawdza czy dany obiekt istnieje, po czym powstaje wektor odpowiedzialny za reprezentację kierunku. Dalsze linie kodu odpowiedzialne są za obracanie względem osi Y o 90 stopni. Wartości zostały dostosowane do modelu, który został wykorzystany w projekcie, dlatego mogą być one różne w zależności od modelu. Po upewnieniu się że wartość wektora jest nierówna zeru następuje obrót obiektu, funkcja „Slerp” odpowiedzialna jest tutaj za płynny obrót rozłożony w czasie. W przypadku jej braku nastąpiłby gwałtowny obrót, zajmujący jedną klatkę.

Aby pion rozpoczął podążanie za graczem musi on znajdować się w zasięgu jego widzenia za co odpowiedzialna jest funkcja zwracająca wartość boolowską „PlayerInSight” widoczna na listingu 15.3.

```
private bool PlayerInSight()
{
    if (player != null)
    {
        float playerX = player.position.x;
        float pawnX = rb.position.x;
        float playerY = player.position.y;
        float pawnY = rb.position.y;
        if (Mathf.Abs(playerX - pawnX) < areaOfSight && Mathf.Abs(playerY - pawnY) < areaOfSight)
        {
            return true;
        }
    }
    return false;
}
```

*Listing 15.3. Metoda PlayerInSight*

Funkcja sprawdza czy wartość bezwzględna różnicy współrzędnych gracza do obiektu jest mniejsza niż zasięg widzenia danego przeciwnika, jeżeli tak oznacza to że gracz znajduje się w zasięgu widzenia a zwracana wartość równa jest „true”, w przeciwnym wypadku zwracana jest wartość false. Funkcja ta zostaje wywołana w metodzie „Update” (listing 15.4.). Po zwróceniu wartości „true”, przeciwnik rozpoczyna podążanie za graczem.

```

void Update()
{
    if (isAlive)
    {
        if (cooldownAttack)
        {
            attackCooldownTimer -= Time.deltaTime;
            if (attackCooldownTimer <= 0f)
            {
                cooldownAttack = false;
            }
        }

        if (!cooldownAttack)
        {
            if (Vector3.Distance(player.position, rb.position) < attackRange)
            {
                UnityEngine.Debug.Log("Attack");
                animator.SetTrigger("Attack");
                cooldownAttack = true;
                attackCooldownTimer = 1f / attackSpeed;
            }
        }
    }

    if (PlayerInSight() && player != null)
    {
        ChasingPlayer();
        LookAtPlayer();
    }
    else
    {
        MovingBetweenPoints();
    }
}
}

```

*Listing 15.4. Mechanizm opóźnienia ataku*

#### **Podążanie za graczem:**

Za podążanie za graczem odpowiedzialna jest metoda „ChasingPlayer” zaprezentowana na listingu 15.5. Metoda zapisuje współrzędne gracza a następnie oblicza dystans, pomiędzy graczem a obiektem.

```

private void ChasingPlayer()
{
    Vector3 target = new Vector3(player.position.x, rb.position.y, rb.position.z);
    float distanceToPlayer = Vector3.Distance(rb.position, target);

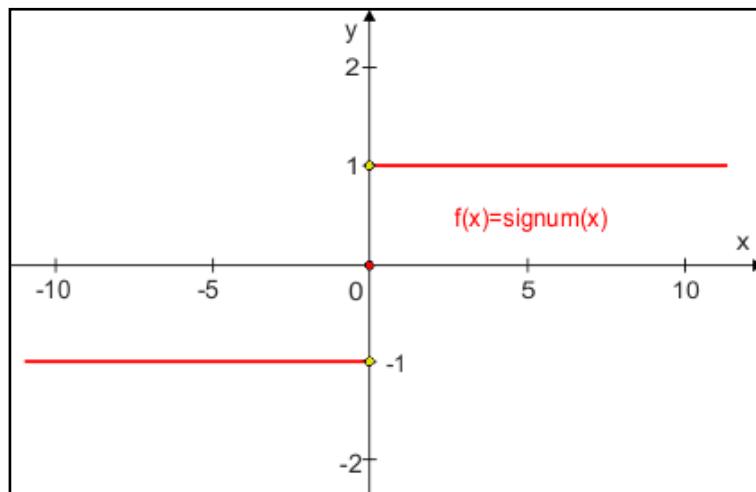
    if (distanceToPlayer > 0.1f)
    {
        float direction = Mathf.Sign(target.x - rb.position.x);
        rb.velocity = new Vector3(speed * 1.5f * direction, 0, 0);

        float clampedX = Mathf.Clamp(rb.position.x, pointA.transform.position.x, pointB.transform.position.x);
        rb.position = new Vector3(clampedX, rb.position.y, rb.position.z);
    }
}

```

*Listing 15.5. Metoda ChasingPlayer*

Jeżeli istnieje różnica między współrzędnymi gracza i obiektu, następuje obliczenie kierunku, w którym ma się poruszać obiekt, wykorzystywana w tym celu jest funkcja „Sign” z biblioteki „Mathf” jest to funkcja Signum Rys. 15.2.



Rys. 15.2. Wykres funkcji sign

Dzięki danej funkcji zostaje określony kierunek, w którym powinien przemieszczać się obiekt, aby napotkać punkt, w którym znajduje się gracz. W przypadku gdy gracz znajduje się po lewej stronie obiektu, różnica punktu docelowego, a obiektu jest mniejsza od zera, dzięki czemu funkcja „Sign” zwraca wartość „-1”. Natomiast jeżeli gracz znajduje się po prawej stronie obiektu zwracana jest wartość „1”. Dzięki temu rozwiązaniu możliwe jest pomnożenie zmiennej „X” wektora prędkości o zmienną „direction”. Funkcja „Sign” służy tutaj za zabezpieczenie przed niepożądaną zmianą prędkości. Dzięki funkcji „Clamp” możliwe jest zablokowanie podążania za graczem poza punkty A i B. Następnie zostają przypisane nowe wartości dla wektora, wraz z zablokowanymi osiami Y, Z oraz ograniczeniem w postaci punktów A oraz B.

```
private void LookAtPlayer()
{
    if (player != null)
    {
        Vector3 lookDirection = player.position - transform.position;
        if (lookDirection.x < 0)
        {
            rb.transform.rotation = Quaternion.Euler(0f, 0f, 0f);
        }
        else
        {
            rb.transform.rotation = Quaternion.Euler(0f, 180f, 0f);
        }
    }
}
```

Listing 15.6. Metoda LookAtPlayer

Tak jak w przypadku podążania pomiędzy punktami, wymagane jest obsłużenie obracania się obiektu w stronę gracza. W celu rozwiązania tego problemu powstała metoda „LookAtPlayer” patrz listing 15.6., która zostaje wywołana w metodzie „Update” wraz z metodą „ChasingPlayer”. Poprzez obliczenie różnicy w dystansie pomiędzy graczem a obiektem zostaje określone ich położenie względem siebie, następnie bazując na tej wartości, rotacja obiektu zostaje dostosowana o 180 stopni względem osi Y. Umożliwia to zwrot obiektu w kierunku gracza.

### Atakowanie gracza:

Kod odpowiedzialny za wywołanie animacji ataku maczugą przez piona znajduje się w metodzie „Update”, dzięki czemu czas pomiędzy atakami jest ciągle obliczany. Kod odpowiedzialny za atak został zaprezentowany na listingu nr 15.7.

```
if (cooldownAttack)
{
    attackCooldownTimer -= Time.deltaTime;
    if (attackCooldownTimer <= 0f)
    {
        cooldownAttack = false;
    }
}

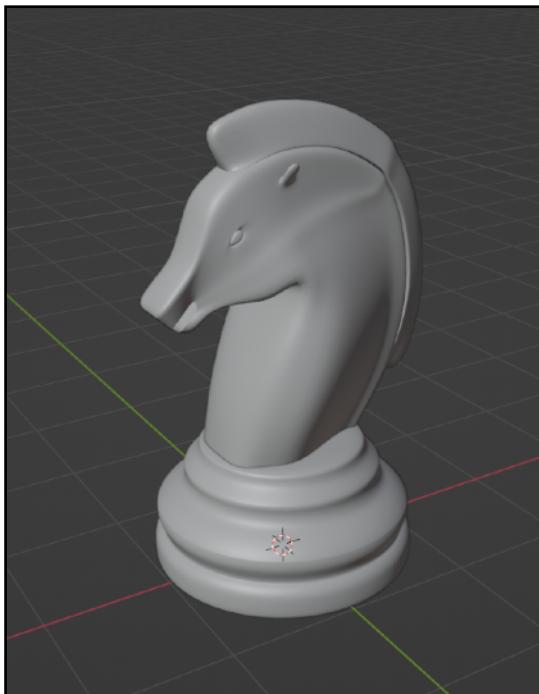
if (!cooldownAttack)
{
    if (Vector3.Distance(player.position, rb.position) < attackRange)
    {
        UnityEngine.Debug.Log("Attack");
        animator.SetTrigger("Attack");
        cooldownAttack = true;
        attackCooldownTimer = 1f / attackSpeed;
    }
}
```

Listing 15.7. Fragment kodu odpowiedzialnego za atak

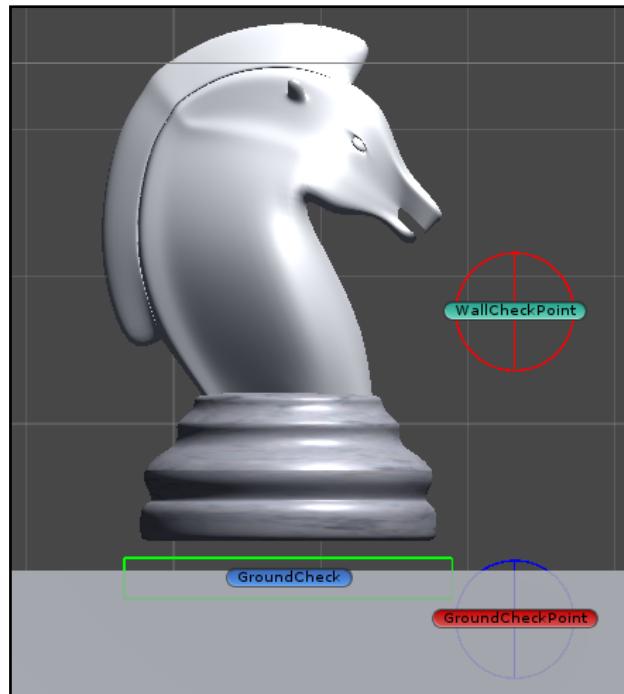
Powyższy fragment kodu oblicza czas do następnego ataku, a następnie gdy czas wymagany do kolejnego ataku wyniesie „0” zostaje sprawdzony dystans pomiędzy graczem i obiektem. Jeżeli dystans ten jest mniejszy od zasięgu ataku, następuje wywołanie ataku poprzez przesłanie wyzwalacza do animatora, który jest odpowiedzialny za animacje obiektu. Następnie czas potrzebny do aktywowania kolejnego ataku zostaje zresetowany, tak samo jak i wartość zmiennej boolowskiej reprezentującej koniec czasu niezbędnego do wywołania animacji.

### 15.1.2. Biały Skoczek

Do wykonania szczegółowych elementów modelu wykorzystano dostępne w tym programie narzędzia „extrude” oraz „intersect”. Model przedstawiono na rys. 15.3.



Rys. 15.3. Model białego skoczka



Rys. 15.4. Model Bialego skoczka z nalożonymi obiektymi do wykrywania przeszkód

#### Poruszanie się:

Do utworzenia „patrolowania” przez Skoczka skorzystano z innego podejścia niż w przypadku Piona, poruszanie się między wcześniej określonymi punktami zostało zastąpione na poruszanie się do momentu napotkania przeszkody lub braku możliwości dalszego poruszania się. Aby umożliwić wykrycie możliwych przeszkód w poruszaniu się do obiektu, dodano puste obiekty: „GroundCheck”, „WallCheckPoint” oraz „GroundCheckPoint” co zostało zaprezentowane na rys 15.4. Dzięki relacji „Parent – Children” punkty przemieszczają się wraz z przeciwnikiem.

```
void FixedUpdate()
{
    Collider[] groundColliders = Physics.OverlapSphere(groundCheckPoint.position, sphereRadius, groundLayer);
    checkingGround = groundColliders.Length > 0;

    Collider[] wallColliders = Physics.OverlapSphere(wallCheckPoint.position, sphereRadius, groundLayer);
    checkingWall = wallColliders.Length > 0;

    Collider[] isgroundColliders = Physics.OverlapBox(groundCheck.position, boxSize / 2f, Quaternion.identity, groundLayer);
    isGrounded = isgroundColliders.Length > 0;
}
```

Listing 15.8. Metody wykorzystujące obiekty do wykrywania przeszkód

Wykorzystanie tych obiektów w kodzie zaprezentowano na listingu x w metodzie „FixedUpdated”. Metoda „FixedUpdated” jest to metoda odpowiedzialna za fizykę obiektów, jednak może generować problemy w przypadkach związanych z odczytem z klawiatury. W takim przypadku do odczytów powinno się zastosować metodę „Update”, ponieważ jest wykonywana w każdej klatce programu, a do operacji fizycznych na danym

obiekcie „ FixedUpdate” gdyż wykonywana jest co stały krok czasowy niezależny od wydajności komputera .

Kod podany w listingu 15.9. określa czy dany punkt ma kontakt z ziemią, ścianą lub powierzchnią o określonej warstwie. Jeżeli tak, zwrócona zostaje wartość „true” do odpowiednich parametrów „ checkingGround”, „ checkingWall” oraz „ isGrounded”, które zostaną wykorzystane podczas poruszania się.

```
void FixedUpdate()
{
    Collider[] groundColliders = Physics.OverlapSphere(groundCheckPoint.position, sphereRadius, groundLayer);
    checkingGround = groundColliders.Length > 0;

    Collider[] wallColliders = Physics.OverlapSphere(wallCheckPoint.position, sphereRadius, groundLayer);
    checkingWall = wallColliders.Length > 0;

    Collider[] isGroundColliders = Physics.OverlapBox(groundCheck.position, boxSize / 2f, Quaternion.identity, groundLayer);
    isGrounded = isGroundColliders.Length > 0;
    if (isAlive)
    {
        if (!PlayerInSight() && isGrounded)
        {

            Patrolling();
        }
        if (cooldownAttack)
        {
            attackCooldownTimer -= Time.deltaTime;
            if (attackCooldownTimer <= 0f)
            {
                cooldownAttack = false;
            }
        }
        if (!cooldownAttack)
        {

            if (PlayerInSight() && isGrounded)
            {
                JumpAttack();
                FlipTowardsPlayer();
                cooldownAttack = true;
                attackCooldownTimer = 1f / attackSpeed;
            }
        }
    }
}
```

Listing 15.9. Kod sprawdzający kontrakt z przeszkodą

```
private void Patrolling() {
    if (!checkingGround || checkingWall) {
        if (facingRight)
        {
            Flip();
        }
        else if (!facingRight) {
            Flip();
        }
    }
    rb.velocity = new Vector3(speed * moveDirection, rb.velocity.y, 0);
}
```

Listing 15.10. Metoda Patrolling

Metoda „FixedUpdate” jest metodą zbliżoną do zaprezentowanej wcześniej „Update” należącą do „Białego Piona” dlatego też metoda „Patrolling” odpowiedzialna za przemieszczenie się „Białego Skoczka” wywołana zostaje w momencie gdy gracz nie znajduje się w polu widzenia obiektu i gdy dany obiekt znajduje się na ziemi. Metoda „PlayerInSight” w przypadku „Białego Skoczka” jest tą samą, która została zastosowana dla „Białego Piona” (rozdział 15, podpunkt 1.1). W celu zmiany zachowania poruszania się „Białego Skoczka” względem „Białego Piona”, zmieniona została funkcja „Patrolling” zaprezentowana na listingu 15.10. Metoda „Patrolling” jest odpowiedzialna za wykrywanie czy obiekt znalazł się w sytuacji, gdzie musi nastąpić zmiana w poruszaniu się, którą obsługuje metoda „Flip”, zawarta na listingu 15.11.

```
private void Flip() {
    moveDirection *= -1;
    facingRight = !facingRight;
    transform.Rotate(0, 180, 0);
}
```

*Listing 15.11. Metoda Flip*

Metoda „Flip” podczas wywołania zmienia zwrot poruszania się oraz rotację obiektu. Gdy zostanie wywołana w metodzie „Patrolling”, następuje zmiana wektora, a wartość prędkości zostaje zmieniona na przeciwną.

#### **Atakowanie gracza:**

Atakowanie gracza w przypadku „Białego Skoczka” następuje w podobnym przypadku jak „Białego Piona” (rozdział 15, podpunkt 1.1), a odpowiedzialny za to kod znajduje się na listingu 15.12. Podczas ataku, przeciwnik doskakuje w stronę gracza.

```
if (cooldownAttack)
{
    attackCooldownTimer -= Time.deltaTime;
    if (attackCooldownTimer <= 0f)
    {
        cooldownAttack = false;
    }
}
if (!cooldownAttack)
{

    if (PlayerInSight() && isGrounded)
    {
        JumpAttack();
        FlipTowardsPlayer();
        cooldownAttack = true;
        attackCooldownTimer = 1f / attackSpeed;
    }
}
```

*Listing 15.12. Kod odpowiedzialny za atak białego skoczka*

Jeżeli gracz znajduje się w zasięgu ataku obiektu, wartość „isGrounded” wynosi „true” oraz wartość zmiennej boolowskiej reprezentującej koniec czasu potrzebnego do wykonania ataku wynosi „false”, następuje atak wywołany metodą „JumpAttack” patrz listing 15.13. Metoda „JumpAttack” oblicza dystans pomiędzy graczem i obiektem, jeżeli dany obiekt znajduje się na ziemi, zostaje dodany do niego wektor siły wraz z modyfikatorem „Impulse”, który oznacza że siła powinna zostać dodana gwałtownie i przez krótki okres czasu.

```
private void JumpAttack()
{
    float distanceFromPlayer = player.position.x - transform.position.x;
    if (isGrounded)
    {
        rb.AddForce(new Vector2(distanceFromPlayer, jumpHeight), ForceMode.Impulse);
    }
}
```

*Listing 15.13. Metoda JumpAttack*

Metoda „FlipTowardsPlayer” jest metodą utworzoną na potrzeby „Białego Piona”, która również została wykorzystana w implementacji „Białego Skoczka” oraz została ona już opisana (rozdział 15, podpunkt 1.1) .

### *15.1.3. Biały Goniec*

Model „Białego Skoczka” został zaprezentowany na rys. Obiekt utworzono w programie do grafiki trójwymiarowej Blender (rys. 15.5.). Utworzenie animacji zostało wykonane w Unity z zastosowaniem narzędzia animator.



*Rys. 15.5. Model Białego Gońca*

## Przemieszczanie się:

Za przemieszczanie się po przekątnej między punktami „Białego Gońca” odpowiada kod zawarty w metodzie „Update” (listing 15.14.). Jest to kod zbliżony do metody „MovingBetweenPoint” „Białego Piona” (rozdział 15, podpunkt 1.1), jednak w tym przypadku zmiana odbywa się w dwóch wymiarach - Osi X oraz osi Y, dzięki czemu obiekt przemieszcza się w stronę wcześniejszych przypisanych mu punktów.

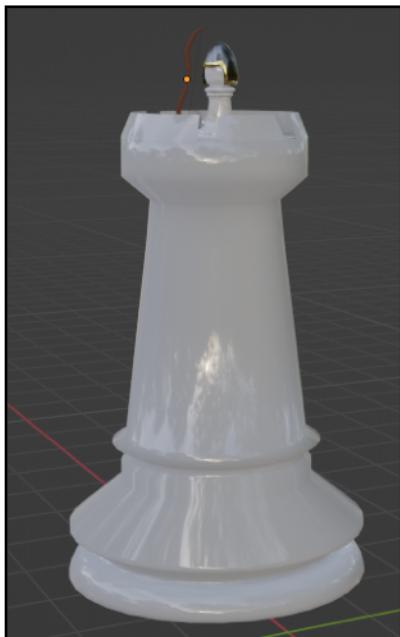
```
void Update()
{
    if (isAlive)
    {
        Vector3 direction = (currentPoint.position - transform.position).normalized;
        rb.velocity = new Vector3(direction.x * speed, direction.y * speed, 0);

        if (Vector3.Distance(transform.position, currentPoint.position) < 0.5f)
        {
            if (currentPoint == pointA)
            {
                currentPoint = pointB;
            }
            else
            {
                currentPoint = pointA;
            }
        }
    }
}
```

Listing 15.14. Fragment skryptu poruszania się białego gońca

#### 15.1.4. Biała Wieża

Model „Białej wieży” został przedstawiony na rys. Obiekt utworzono w programie do grafiki trójwymiarowej Blender (rys. 15.6.). Jako postać strzelca, została wykorzystany model „Białego Piona”.

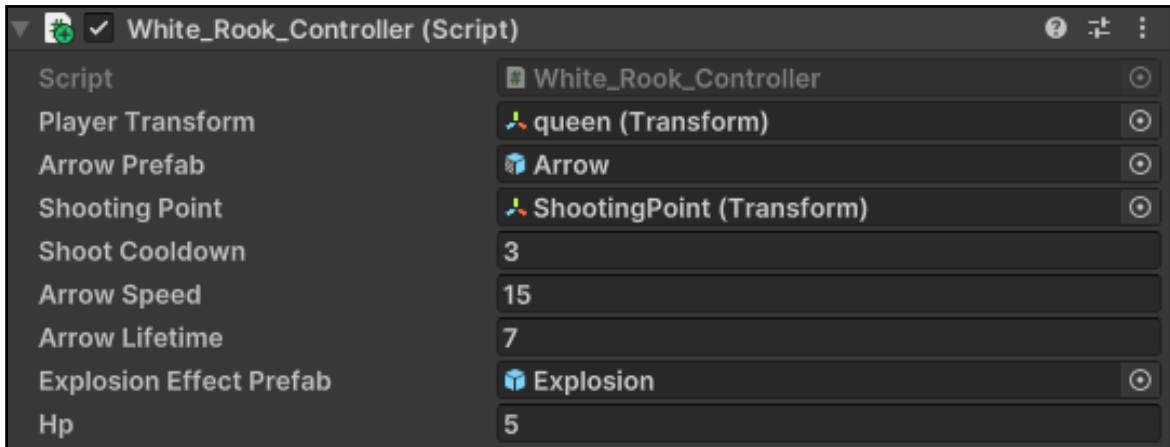


Rys 15.7. Biała wieża atakująca gracza

Rys 15.6. Model Białej Wieży

Wieża jest nieruchomym obiektem, który wykonuje atak w momencie znalezienia się w polu widzenia gracza. „Biała Wieża” wystrzeli wtedy strzałę w kierunku gracza. Model wieży jest zawsze zwrócony w kierunku gracza. Strzała jest tworzona na podstawie wcześniej utworzonego wzorca, z określonymi zależnościami tj.: „Rigidbody” z ustawioną wartością „Use Gravity” na „false” oraz „BoxCollider” z opcją „IsTrigger”. Takie ustawienia obsługują odpowiednie przemieszczanie się strzały oraz zniszczenie jej w momencie kolizji z graczem lub ziemią. Rysunek 15.7. przedstawia Białą Wieżę atakującą gracza.

Za ustawienie właściwości „Białej Wieży” odpowiada klasa „White\_Rook\_Controller”, za pomocą publicznych pól przedstawionych na rysunku 15.8. Dzięki nim można przypisać cel strzału, „prefab” pocisku (wzorzec zapisany w odpowiednim folderze projektu), punkt wystrzału, czas pomiędzy wystrzałami, prędkość pocisku, czas po którym strzała zostanie zniszczona, zdrowie przeciwnika wzorzec animacji eksplozji, wyświetlanej po pokonaniu „Białej Wieży”.



Rys. 15.8. Właściwości Białej Wieży

Naniesiony na przeciwnika skrypt w każdej klatce gry sprawdza, czy należy wywołać metodę odpowiedzialną za wystrzał lub obrót w stronę gracza. Odpowiada za to metoda „Update”, sprawdzająca czy przeciwnik jest gotowy do strzału oraz metoda „FlipTowardsPlayer”, która obraca „Białą Wieżę” w stronę gracza (metody przedstawione na listingu 15.15).

```

void Update()
{
    if (playerTransform == null)
    {
        UnityEngine.Debug.LogError("PlayerTransform not assigned in the inspector!");
        return;
    }

    FlipTowardsPlayer();

    if (readyToShoot)
    {
        Shoot();
        StartCoroutine(ResetShootCooldown());
    }
}

void FlipTowardsPlayer()
{
    float directionToPlayer = playerTransform.position.x - transform.position.x;

    if (directionToPlayer < 0)
    {
        transform.localScale = new Vector3(1.3f, 1.3f, 1.3f);
    }
    else if (directionToPlayer > 0)
    {
        transform.localScale = new Vector3(1.3f, 1.3f, -1.3f);
    }
}

```

Listing 15.15 Skrypt namierzania gracza przez wieżę

Za wystrzał pocisku odpowiada metoda „Shoot”. Strzała jest tworzona na podstawie wzorca w odpowiednio przypisanej pozycji wystrzału. W celu zachowania odpowiedniego zwrotu strzały, jej skala według osi Z zostaje przemnożona przez obecną skalę gracza (jej wartość na osi Z wynosi -1, jeżeli gracz jest odwrócony). Kierunek wystrzału jest ustalany na podstawie pozycji gracza. Dodatkowo zastosowano wektor przesuwający cel w górę, ponieważ domyślna pozycja gracza znajduje się przy stopach postaci. Następnie strzała otrzymuje odpowiednią siłę wystrzału oraz prędkość, a po upłynięciu odpowiedniego czasu jej obiekt zostanie zniszczony.

```
void Shoot()
{
    GameObject arrow = Instantiate(arrowPrefab, shootingPoint.position, shootingPoint.rotation);
    Rigidbody arrowRb = arrow.GetComponent<Rigidbody>();

    arrow.transform.localScale = new Vector3(arrow.transform.localScale.x, arrow.transform.localScale.y,
                                             arrow.transform.localScale.z * transform.localScale.z);

    Vector3 shootDirection = (playerTransform.position + new Vector3(0f, 3f, 0f) - shootingPoint.position).normalized;
    arrowRb.AddForce(shootDirection * arrowSpeed, ForceMode.Impulse);

    Destroy(arrow, arrowLifetime);
}
```

*Listing 15.16. Metoda Shoot*

Skrypt klasy „Arrow” (fragment przedstawiono na listingu 15.17.) jest przypisany do wzorca strzały, tak aby każda wystrzelona strzała przez Białą Wieżę mogła z niego korzystać.

```
void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player") || other.CompareTag("Ground") || other.CompareTag("Projectile"))
    {
        Destroy(gameObject);
    }
}

void Update()
{
    CheckIfOutOfScreen();
}

void CheckIfOutOfScreen()
{
    Vector3 viewportPoint = mainCamera.WorldToViewportPoint(transform.position);

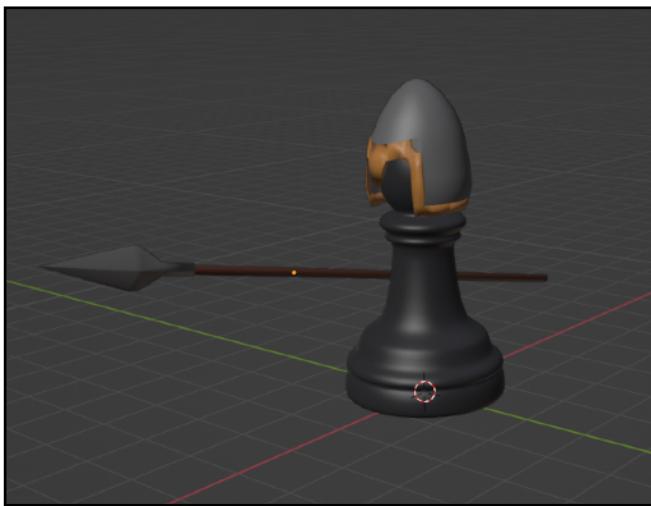
    if (viewportPoint.x < 0 || viewportPoint.x > 1 || viewportPoint.y < 0 || viewportPoint.y > 1)
    {
        Destroy(gameObject);
    }
}
```

*Listing 15.17. Fragment klasy Arrow*

Skrypt przedstawiony powyżej posiada trzy procedury do niszczenia strzały w określonych przypadkach: podczas wchodzenia w kolizję z określonymi elementami lub gdy strzała znajdzie się poza obszarem kamery.

### *15.1.5. Czarny Pion*

Model „Czarnego Piona” został przedstawiony na Rys. 15.9. Obiekt utworzono w programie do grafiki trójwymiarowej Blender. Utworzenie animacji zostało wykonane w Unity z zastosowaniem narzędzia animator.

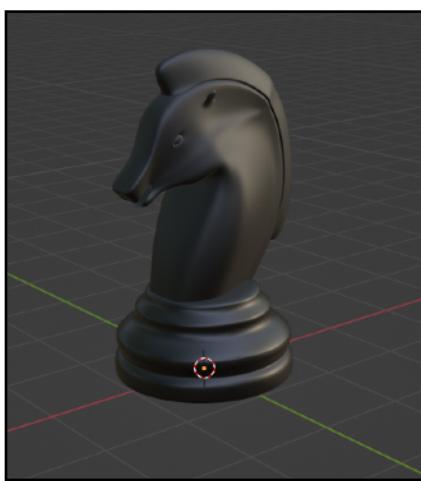


Rys. 15.9. Model czarnego piona

Zachowania „Czarnego Piona” są zbliżone do „Białego Piona” (rozdział 15, podpunkt 1.1) dlatego też został zastosowany ten sam kod, jednak zmienione zostały animacje ataku oraz podstawowe statystyki.

### *15.1.6. Czarny Skoczek*

Model „Czarnego Skoczka” został przedstawiony na Rys. 15.10. Obiekt utworzono w programie do grafiki trójwymiarowej Blender. Do wykonania szczegółowych elementów modelu wykorzystano dostępne w tym programie narzędzia „extrude” oraz „intersect”.



Rys. 15.10. Model czarnego skoczka

Zachowania „Czarnego Skoczka” są zbliżone do „Białego Skoczka” (rozdział 15, podpunkt 1.2) dlatego też został zastosowany ten sam kod, jednak zmienione zostały podstawowe statystyki co spowodowało zmianę w skoku.

### 15.1.7. Czarny Goniec:

Model „Czarnego Gońca” został zaprezentowany na Rys. 15.11. Obiekt utworzono w programie do grafiki trójwymiarowej Blender. Utworzenie animacji zostało wykonane w Unity z zastosowaniem narzędzia animator.



Rys. 15.11. Model Czarnego Gońca

#### Poruszanie się:

„Czarny Goniec” posiada dwie możliwości poruszania się, w momencie poruszania się poza kamerą gracza, obiekt porusza się między dwoma punktami, za co odpowiada kod na listingu 15.18. Jest to kod zbliżony do poruszania się „Białego Gońca” (rozdział 15, podpunkt 1.3).

```
void Patrolling() {
    isPatrolling = true;
    Vector3 direction = (currentPoint.position - transform.position).normalized;

    rb.velocity = new Vector3(direction.x * speed, direction.y * speed, 0);

    if (Vector3.Distance(transform.position, currentPoint.position) < 1f)
    {
        if (currentPoint == pointA)
        {
            currentPoint = pointB;
        }
        else
        {
            currentPoint = pointA;
        }
    }
}
```

Listing 15.18. Metoda Patrolling czarnego gońca

Gdy „Czarny Goniec” znajdzie się w obszarze kamery gracza rozpoczyna poruszać się w tym obszarze bez możliwości opuszczenia go. W celu sprawdzenia czy obiekt znajduje się w obszarze kamery powstała metoda „EnemyBetweenCamera” zaprezentowana na listingu 15.19.

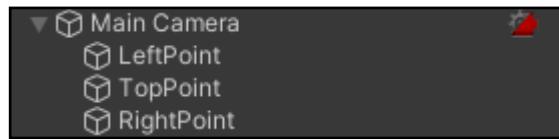
```

private bool EnemyBetweenCamera() {
    if (player != null)
    {
        float enemyX = rb.position.x;
        float enemyY = rb.position.y;
        float leftbarier = leftPoint.position.x;
        float rightbarier = rightPoint.position.x;
        float topbarier = topPoint.position.y;
        if (enemyX > leftbarier && enemyY < topbarier && enemyX < rightbarier)
        {
            return true;
        }
    }
    return false;
}

```

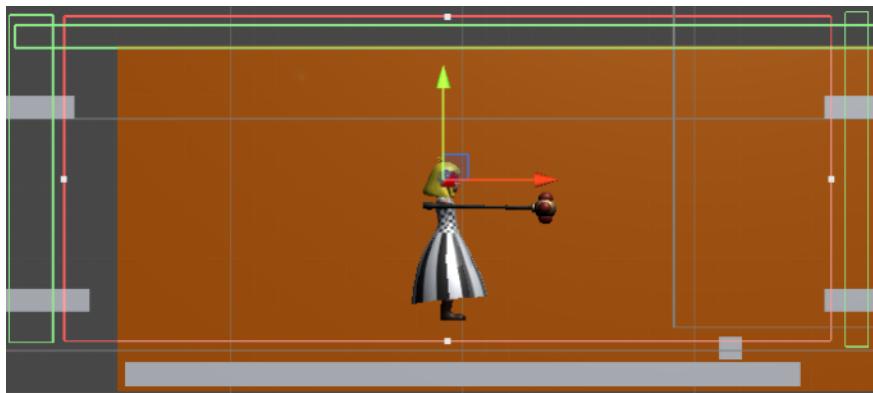
*Listing 15.19. Metoda EnemyBetweenCamera*

Metoda zapisuje współrzędne obiektu i porównuje je do punktów umieszczonych w kamerze gracza w relacji „Parent – Children” (rys. 15.12.), dzięki temu dane punkty będą przemieszczać się wraz z kamerą.



*Rys 15.12.*

Na rysunku 15.13. zaprezentowano podgląd z poziomu testowego gdzie zielonymi obszarami zostały zaznaczone „Box Collidery” poszczególnych pustych obiektów a czerwonym obszar kamery gracza. Dzięki użyciu opcji „IsTrigger” na „Box Colliderze”, możliwym jest przeniknięcie przez niego. W innym przypadku powstałyby nieprzenikalna bariera dla wszelkich obiektów. Zastosowanie „Box Colliderów” umożliwia stworzenie granic dla poruszania się „Czarnego Gońca”.



*Rys. 15.13. Podgląd sceny testowej*

Kod listingu 15.20. odpowiada za wywołanie metod odpowiedzialnych za odpowiednie przemieszczenie. Spełnienie warunków znajdujących się w metodzie „ FixedUpdate” spowoduje poruszanie się między punktami w sposób w jaki wykonywał to „Biały Goniec”. Gdy metoda „EnemyBetweenCamera” zwróci wartość „true”, co będzie

jednoznaczne ze znajdowaniem się obiektu w kamerze gracza, następuje zmiana trybu poruszania.

```
0 references
void Update()
{
    if (EnemyBetweenCamera()) { isPatrolling = false; HitLogic(); }
}

0 references
void FixedUpdate()
{
    if (!EnemyBetweenCamera() && !PlayerInSight()) { Patrolling(); }
    if (!isPatrolling) { rb.velocity = moveDirection * speed; }
}
```

Listing 15.20. Metoda FixedUpdate

Za tryb poruszania się między kamerą odpowiedzialna jest metoda „HitLogic” zaprezentowana na listingu 15.21.

```
void HitLogic()
{
    touchedRight = HitDetector(rightCheck, rightCheckSize, groundLayer | cameraborder);
    touchedLeft = HitDetector(leftCheck, leftCheckSize, groundLayer | cameraborder);
    touchedRoof = HitDetector(roofCheck, roofCheckSize, groundLayer | cameraborder);
    touchedGround = HitDetector(groundCheck, groundCheckSize, groundLayer | cameraborder);
    if (touchedLeft)
    {
        Flip();
    }
    if (touchedRight)
    {
        Flip();
    }
    if (touchedRoof & goingUp)
    {
        ChangeYDirection();
    }
    if (touchedGround && !goingUp)
    {
        ChangeYDirection();
    }
}
```

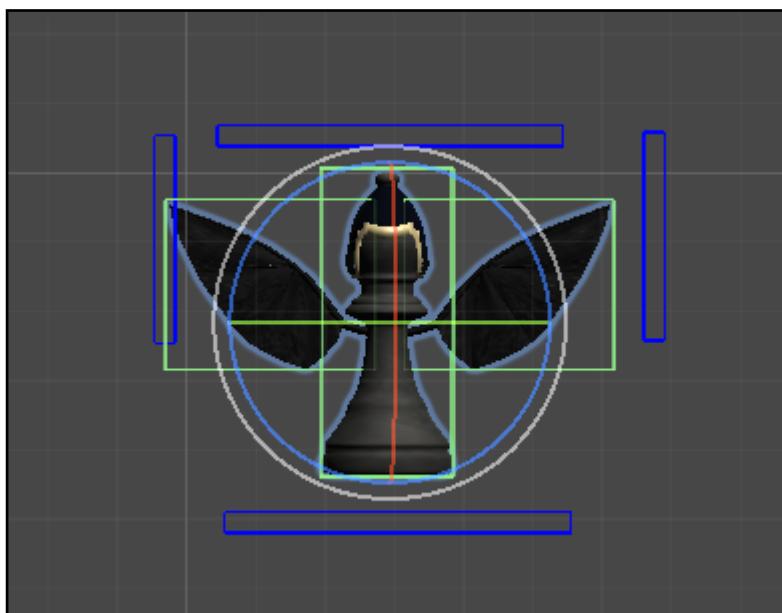
Listing 15.21. Metoda HitLogic

Dzięki metodzie „HitDetector” (Listing 15.22.) sprawdzane jest czy dany obiekt nie wszedł w kolizję z obiektem o odpowiednim oznaczeniu. Po wejściu w kolizje zwrócona zostanie wartość „true”. Parametrami funkcji są: pusty obiekt, rozmiar obiektu oraz jego oznaczenie.

```
4 references
bool HitDetector(GameObject gameObject, Vector3 size, LayerMask layer)
{
    return Physics.OverlapBox(gameObject.transform.position, size / 2, Quaternion.identity, layer).Length > 0;
}
```

Listing 15.22. Metoda HitDetector

Na Rys 15.14. można zaobserwować wielkość obszarów odpowiedzialnych za testowanie kolizji, obszary te są w relacji „Parent - Children”, dzięki czemu przemieszczają się wraz z obiektem.



Rys. 15.14. Model czarnego gońca wraz z obszarami do wykrywania kolizji

W momencie zetknięcia się obszaru sprawdzającego wraz z obiektem posiadającym oznaczenie „groundLayer” bądź „cameraborder”, wartość zwracana przez detektor zmieniana jest na „true”, następnie w zależności od detektora wywoływane są metody zmiany kierunku przemieszczenia.

```
4 references
void Flip()
{
    moveDirection.x = -moveDirection.x;
    Debug.Log("Flipping!");
}
```

Listing 15.23. Metoda Flip

Jeżeli nastąpiło zetknięcie się wraz z obiektem o odpowiednim oznaczeniu z lewej bądź prawej strony przeciwnika zostaje wywołana metoda „Flip”, która jest odpowiedzialna za zmianę kierunku poruszania się po osi X. Metoda została zaprezentowana na listingu 15.23.

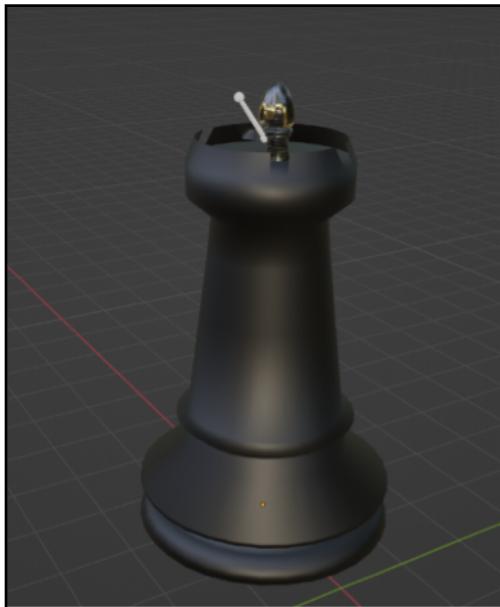
Natomiast gdy zetknięcie nastąpi w górnym bądź dolnym obszarze, wtedy zostaje zmieniona wartość kierunku poruszania się po osi Y na przeciwną (listing 15.24). Wartość flagi „goingUp” jest ustawiana na przeciwną.

```
void ChangeYDirection()
{
    moveDirection.y = -moveDirection.y;
    goingUp = !goingUp;
}
```

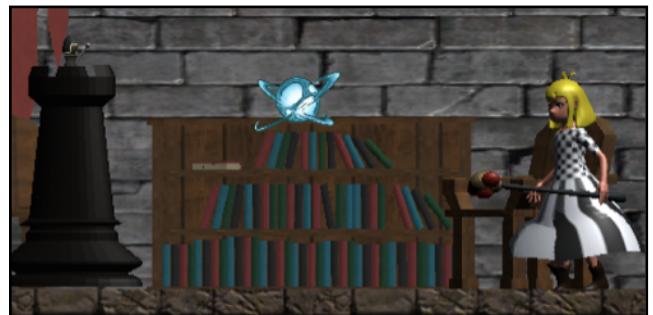
*Listing 15.24. Metoda ChangeYDirection*

#### 15.1.8. Czarna Wieża

Model „Czarnej Wieży” został zaprezentowany na rys. 15.15. Obiekt utworzono w programie do grafiki trójwymiarowej Blender. Jako postać maga, został wykorzystany model „Czarnego Piona”. Jest to wzmacniona wersja „Białej Wieży” (rozdział 15, podpunkt 1.4). Istotną różnicą jest wystrzeliwany pocisk, pozostałe właściwości „Czarnej Wieży” są analogiczne do Białej Wieży. Przeciwnik ten wystrzeliwuje „magiczny pocisk”, który podąża za graczem. Pocisk jest wzorcem animacji, składającej się z czterech obrazków, które są naprzemiennie wyświetlane na ekranie. Rysunek 15.16. przedstawia Czarną Wieżę atakującą gracza.



*Rys. 15.15. Model czarnej wieży*



*Rys. 15.16. Atak czarnej wieży*

Skrypt klasy „MagicMissile” odpowiada za podążanie pocisku za graczem (listing 15.25.). Metoda „Update” pobiera pozycję gracza, uwzględniając przesunięcie celu o określony wektor w góre, aby pocisk podążał w prawidłowy sposób za graczem.

```
void Update()
{
    if (target != null)
    {
        Vector3 targetPosition = target.position + Vector3.up * verticalOffset;

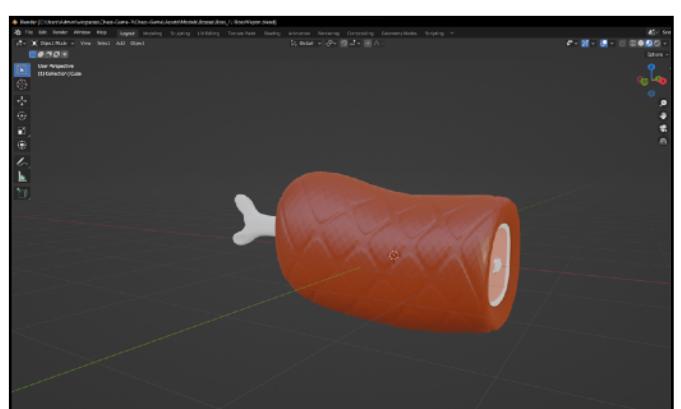
        Vector3 direction = (targetPosition - transform.position).normalized;
        transform.Translate(direction * speed * Time.deltaTime, Space.World);
    }
    else
    {
        transform.Translate(Vector3.right * speed * Time.deltaTime);
    }
}
```

*Listing 15.25. Fragment skryptu MagicMissile*

## 15.3 Implementacja przeciwników specjalnych

### 15.3.1. Leniwy Naczelnik Więzienia

W celu stworzenia Leniwego Naczelnika Więzienia został pobrany darmowy model wraz z animacjami ze strony mixamo.com o nazwie „Castle Guard 02” (rys. 15.17.). Platforma mixamo została opisana w rozdziale o zastosowanych technologiiach (rozdział 11, podpunkt 4).



Rys. 15.18. Model Broni-Szynki

Rys. 15.17. Model naczelnika

Model broni został zaprezentowany na rys. 15.18. Obiekt utworzono w programie do grafiki trójwymiarowej Blender (rozdział 11, podpunkt 3). Tekstury szynki są bazowymi, jednolitymi kolorami z palety barw dostępnej w programie Blender.

#### Przemieszczanie się

Za przemieszczanie się Naczelnika odpowiedzialny jest kod zawarty w listingu 15.26, podany fragment kodu znajduje się w klasie „Boss\_Run”. Metoda „OnStateUpdate” jest funkcją, która uruchamia się w każdej klatce działania aplikacji, dzięki czemu kod w niej zawarty będzie ciągle wywoływany. Powoduje to zczytanie współrzędnych gracza a następnie zmianę pozycji „Naczelnika”, jednak podany kod realizuje jedynie przemieszczenie się „Naczelnika” w kierunku gracza nie zmieniając przy tym jego zwrotu.

```
override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    //Przeszczepienie się w stronę gracza
    Vector3 target = new Vector3(player.position.x, rb.position.y, rb.position.z);
    rb.position = Vector3.MoveTowards(rb.position, target, speed * Time.deltaTime);
    //
```

Listing 15.26. Skrypt poruszania się naczelnika

```

public class Boss_Turning : MonoBehaviour
{
    public Transform player;
    private float distance;
    public bool isFlipped = false;

    2 references
    public void LookAtPlayer()
    {
        if (player != null)
        {
            Vector3 flipped = transform.localScale;
            flipped.z *= -1f;

            if (transform.position.x > player.position.x && !isFlipped)
            {
                transform.localScale = flipped;
                isFlipped = true;
            }
            else if (transform.position.x < player.position.x && isFlipped)
            {
                transform.localScale = flipped;
                isFlipped = false;
            }
        }
    }
}

```

*Listing 15.27. Klasa Boss\_Turning*

Metoda znajdująca się na listingu 15.27. umożliwia zmianę zwrotu w kierunku gracza dzięki testowaniu flagi „isFlipped” oraz pozycji gracza. Gdy gracz znajduje się po lewej stronie „Naczelnika” a flaga „isFlipped” jest ustawiona na „false”, metoda odwraca przeciwnika, zmieniając jego skale na odwrotną. Następnie ustawia flagę „isFlipped” na wartość true, aby pamiętać o odwróceniu obiektu. Analogicznie, gdy gracz znajduje się po prawej stronie „Naczelnika” a flaga „isFlipped” jest ustawiona na wartość „true”, metoda zmienia skalę obiektu, czym odwraca „Naczelnika” w stronę gracza. Następnie ustawia wartość flagi „isFlipped” na „false”, aby pamiętać, że obiekt nie jest teraz odwrócony.

```

override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    boss_turning.LookAtPlayer();
    //Przesunięcie się w stronę gracza
    Vector3 target = new Vector3(player.position.x, rb.position.y, rb.position.z);
    rb.position = Vector3.MoveTowards(rb.position, target, speed * Time.deltaTime);
    //
}

```

*Listing 15.28. Fragment skryptu Boss\_Run*

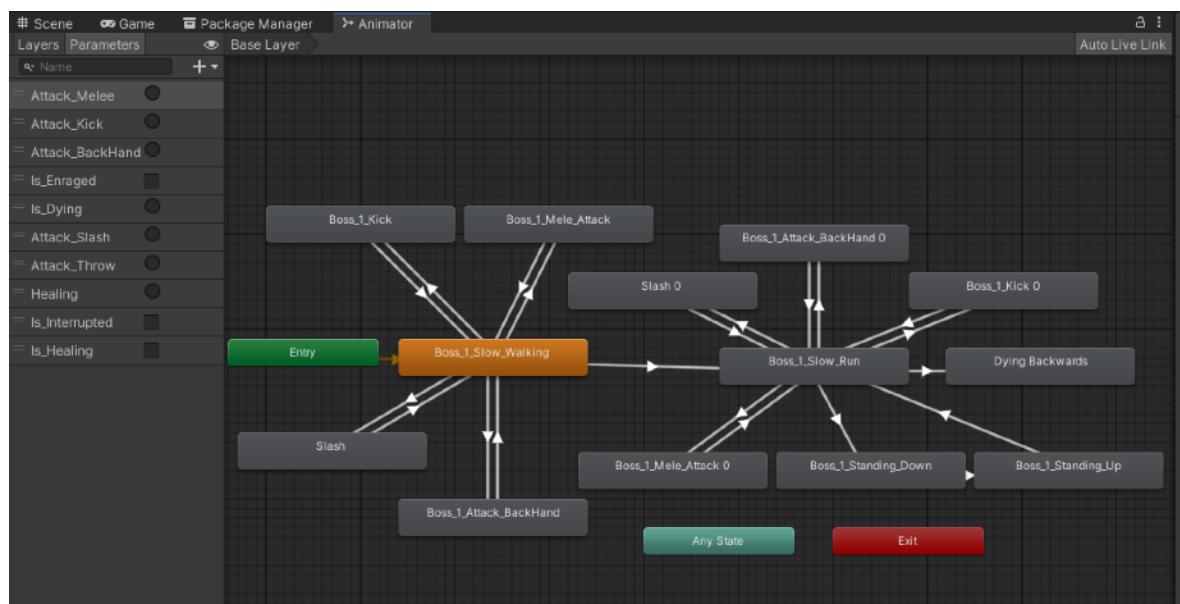
Następnie metoda została wywołana w klasie „Boss\_Run” co zostało zaprezentowane na listingu 15.28.

Aby umożliwić podążanie za graczem wraz ze zwrotem w jego kierunku, powstała metoda „LookAtPlayer” zawarta w pliku „Boss\_Turning”. Metoda znajdująca się na listingu 15.29. umożliwia zmianę zwrotu w kierunku gracza dzięki testowaniu flagi „isFlipped” oraz pozycji gracza.

### Podstawowe ataki:

W celu utworzenia ataków zastosowane zostało wbudowane narzędzie w Unity – Animator

Jest to narzędzie pozwalające na dodawanie animacji do danych obiektów. Na rys. 15.19 pokazany został finalny wygląd animatora „Naczelnika”.



Rys. 15.19. Animator Naczelnika

### Zestaw ruchów:

#### Faza 1:

Przemieszczenie:

- „Naczelnik” porusza się idąc w stronę gracza.

Podstawowe ataki:

- Cios szynką
- Kopnięcie
- Szybkie cięcie szynką
- Cięcie od dołu szynką

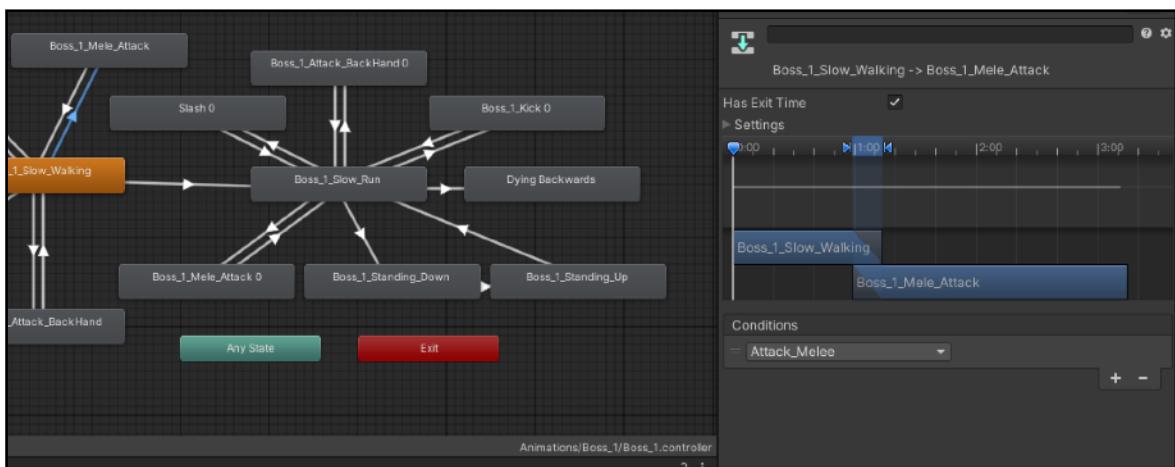
**Przemieszczenie** – „Naczelnik” porusza się idąc w stronę gracza. Jest to pierwsza animacja wywołana po wejściu na arenę bossa i jest podstawową animacją, która jest wywoływana w sposób ciągły, dopóki nie zostaną spełnione warunki do wykonania innej animacji. Przykład takich warunków można zaobserwować na Rys. 15.20. gdzie

wyzwalacze oznaczone zostały kołami a zmienne boolowskie – kwadratami. Gdy zostanie zmieniona wartość boolowska bądź uruchomiony zostanie wyzwalacz, odpowiednia figura geometryczna zostanie zaznaczona tak jak w przypadku Attack\_Melee. Następnie zostaną sprawdzone warunki do wykonania animacji.



Rys.15.20. Zmienne wykorzystywane przez Animatora

W celu ustawienia warunków wymaganych do uruchomienia animacji należy utworzyć „Transition” i połączyć ze sobą dwie animacje znajdujące się w okienku animatora co zostało zaprezentowane na rys 15.21.



Rys 15.21. Animacje naczelnika

Dana konfiguracja umożliwia uruchomienie animacji ręcznie w animatorze. W celu automatycznego uruchomienia animacji, uwzględniono odpowiednie funkcje.

## Podstawowe ataki:

Do obsługi podstawowych ataków zostały napisane funkcje z listingów od 15.29. do 15.31. W celu obsługi ataków został zmodyfikowany kod z skryptu „Boss\_Run”. W funkcji „OnStateUpdate” (15.29.) obliczany zostaje dystans pomiędzy graczem, a „Naczelnikiem”. Gdy gracz znajdzie się w zasięgu ataku „Naczelnika”, ten wywołuje funkcję odpowiedzialną za atak. Zmienna epsilon jest wykorzystywana jako margines błędu, dzięki czemu wartości zbliżone do „attackRange” będą również rozpatrywane, wprowadza ona korekty związane z niedoskonałością arytmetyki zmienoprzecinkowej.

```
0 references
override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    boss_turning.LookAtPlayer();
    //Przemieszczenie się w stronę gracza
    Vector3 target = new Vector3(player.position.x, rb.position.y, rb.position.z);
    rb.position = Vector3.MoveTowards(rb.position, target, speed * Time.deltaTime);
    //
    float distance = Vector3.Distance(player.position, rb.position);
    float epsilon = 0.01f;

    if (distance <= attackRange + epsilon)
    {
        Attack(animator);
    }
}
```

Listing 15.29. Obliczanie dystansu i przygotowanie ataku

```
private bool isAttacking = false;

1 reference
public void Attack(Animator animator)
{
    if (!isAttacking)
    {
        MonoBehaviour monoBehaviour = animator.GetComponent<MonoBehaviour>();

        monoBehaviour.StartCoroutine(AttackCoroutine(animator));
    }
}
```

Listing 15.30. Metoda blokująca zbyt szybkie wywołanie ataku

Metoda „Attack” została utworzona w celu ograniczenia wywoływanego animacji ataku do momentu ustawienia wartości „isAttacking” na „false”. Gdy atak jest możliwy, metoda wywołuje interfejs odpowiedzialny za atak zaprezentowany na listingu 15.30.

Interfejs „AttackCoroutine” (listing 15.31.) ustawia wartość flagi „isAttacking” na „true”, w celu zablokowania ponownego wywołania interfejsu, zanim ten się wykonie. Następnie za wywołanie animacji ataków odpowiada instrukcja warunkowa switch-case, która aktywuje wyzwalacz odpowiedniej animacji zależnej od wylosowanej losowej liczby.

```

private IEnumerator AttackCoroutine(Animator animator)
{
    isAttacking = true;

    int randomAttack = Random.Range(1, 5);
    Debug.Log(randomAttack);

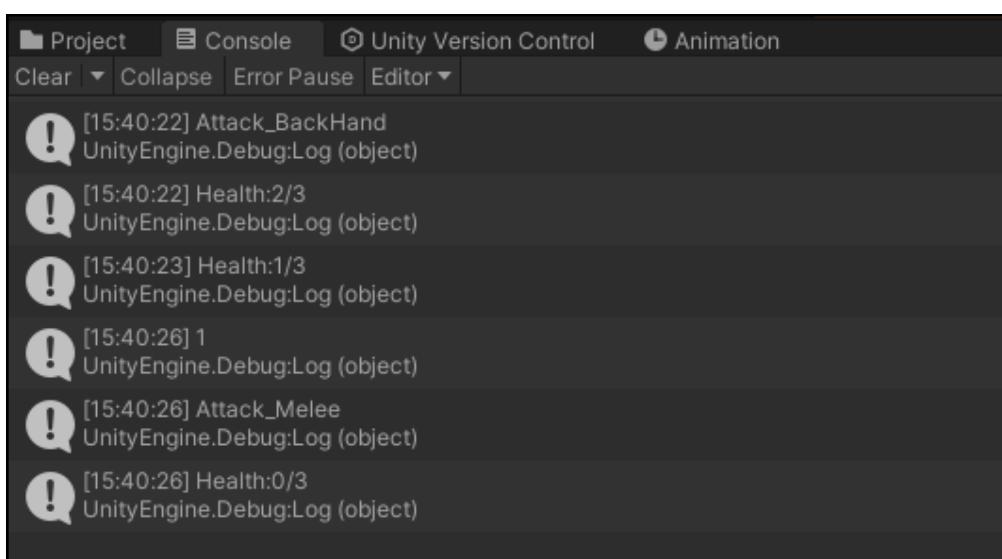
    switch (randomAttack)
    {
        case 1:
            animator.SetTrigger("Attack_Melee");
            Debug.Log("Attack_Melee");
            break;
        case 2:
            animator.SetTrigger("Attack_Kick");
            Debug.Log("Attack_Kick");
            break;
        case 3:
            animator.SetTrigger("Attack_BackHand");
            Debug.Log("Attack_BackHand");
            break;
        case 4:
            animator.SetTrigger("Attack_Slash");
            Debug.Log("Attack_Slash");
            break;
        default:
            break;
    }

    yield return new WaitForSeconds(animator.GetCurrentAnimatorStateInfo(0).length);
    isAttacking = false;
}

```

*Listing 15.31. Metoda odpowiedzialna za atakowanie*

W celu poprawnego wywoływania animacji został zatrzymany współprogram, do momentu aż dana animacja się zakończy. Wtedy wartość flagi „isAttacking” zostaje ustaliona na „false” i umożliwia ona ponowne wykonanie ataku. Dodatkowo zostaje wyświetlona informacja zwrotna w konsoli w środowisku Unity Rys. 15.22.



*Rys. 15.22. Widok komunikatów konsoli podczas atakowania*

Po opuszczeniu stanu animatora, wszystkie wyzwalacze zostają zresetowane. Jest to akcja wykonana w celu uniknięcia niepożądanego wywołania się animacji. Procedura „OnStateExit” (listing 15.32.) jest generowana automatycznie, podczas wygenerowania pliku C#.

```
0 references
public override void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    animator.ResetTrigger("Attack_Kick");
    animator.ResetTrigger("Attack_Melee");
    animator.ResetTrigger("Attack_BackHand");
    animator.ResetTrigger("Attack_Slash");
}
```

Listing 15.32. Metoda zabezpieczająca przed błędami w animacji

### Przejście do fazy 2:

Aby druga faza „Naczelnika” została aktywowana, poziom jego zdrowia musi osiągnąć 50% maksymalnego zdrowia. W tym celu powstała metoda do otrzymywania obrażeń przez „Naczelnika”, zaprezentowana na listingu 8. Funkcja ta nie tylko zarządza otrzymywaniem obrażeń przez „Naczelnika”, ale również odpowiada za przejście do 2 fazy, umiejętność specjalną „Leczenie” oraz śmierć „Naczelnika”.

```
1 reference
public void TakeDamage()
{
    boss_1_health -= 1;
    healthBar.value = boss_1_health;
    if (boss_1_health < bosshalfhp)
    {

        if (animator != null)
        {

            animator.SetBool("Is_Enraged", true);
            boss_1_speed = enragespeed;
        }
    }
    if (boss_1_health <= bossquarterhp && animator.GetBool("Have_Healed") == false )
    {

        animator.SetBool("Is_Healing", true);
    }

    if (boss_1_health <= 0)
    {
        animator.SetTrigger("Is_Dying");
    }
}
```

Listing 15.33. Metoda otrzymywania obrażeń

Wywołanie metody „TakeDamage” (listing 15.33.) odbywa się poprzez procedurę zaprezentowaną na listingu X. Gdy Naczelnik wejdzie w kolizję z obiektem posiadającym oznaczenie „Weapon” oraz w danym momencie jest wykonywana animacja ataku gracza, wywołana zostaje metoda „TakeDamage”.

```

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Weapon") && playerAnimation.isAttacking)
    {
        TakeDamage();
        playerAnimation.isAttacking = false;
        if (animator.GetBool("HealingRightNow") == true) {
            animator.SetBool("Is_Interrupted",true);
        }
    }
}

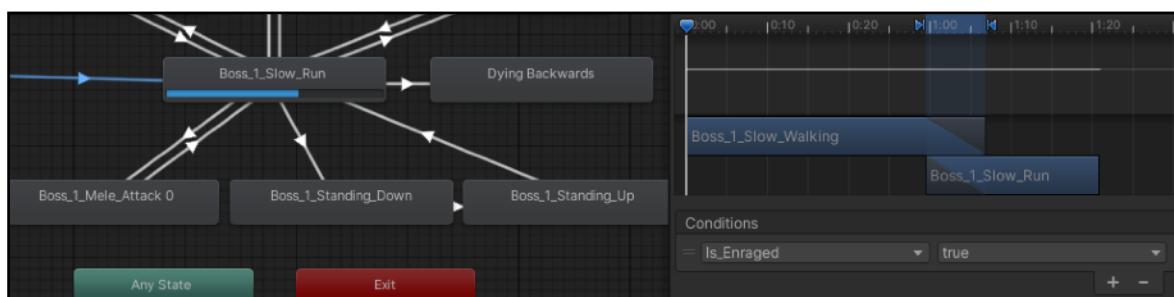
```

*Listing 15.34. Metoda wywołująca otrzymanie obrażeń*

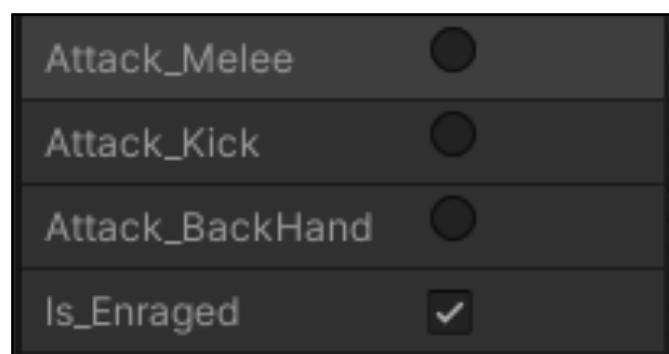
Gdy „Naczelnik” osiągnie 50% swojego maksymalnego zdrowia, zmienna boolowska „Is\_Enraged” w animatorze zostaje ustawiona na wartość „true”, a animacją podstawową bossa zostaje „Boss\_1\_Slow\_Run”. Wtedy też rozpoczyna się druga faza walki z „Naczelnikiem”. Zmiany te pokazują rysunki 15.23. oraz 15.24.

## Faza 2:

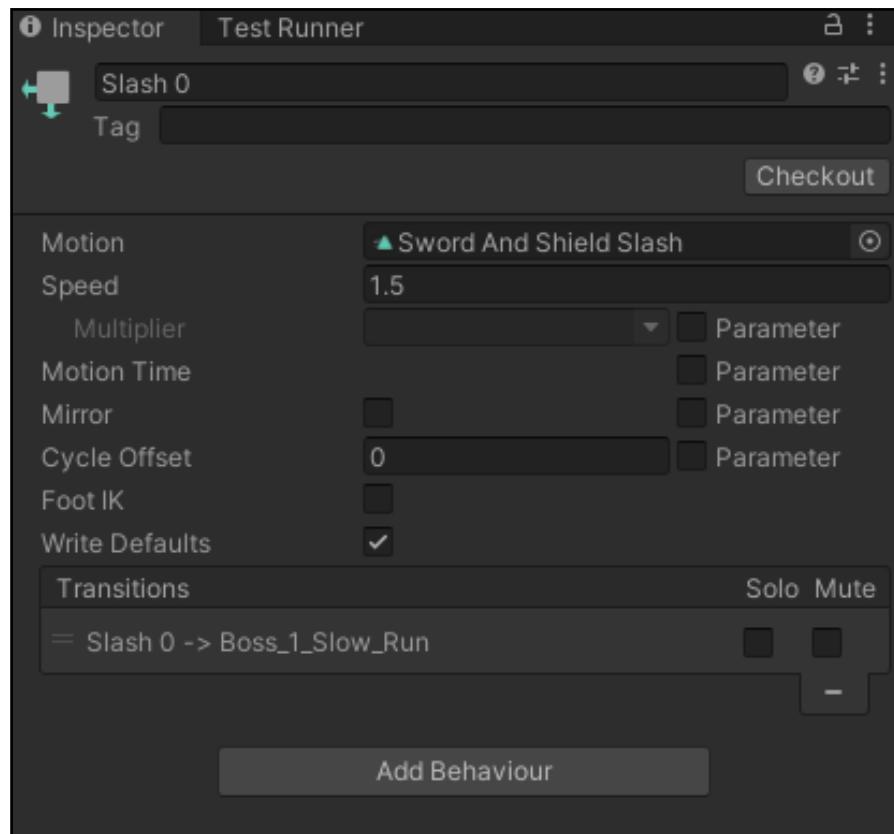
W fazie drugiej „Naczelnik” posiada te same animacje ataków podstawowych oraz ten sam kod obsługujący ich wywołanie. Jednak tempo z jakim wywoływane są animacje jest zwiększone. Ustawienie tego umożliwiaAnimator co zostało zaprezentowane na rys 15.25.



*Rys. 15.23. Animator drugiej fazy walki Naczelnika*



*Rys. 15.24. warunek przejścia do drugiej fazy*



Rys. 15.25. Przykład animacji drugiej fazy Naczelnika

### Umiejętności specjalne:

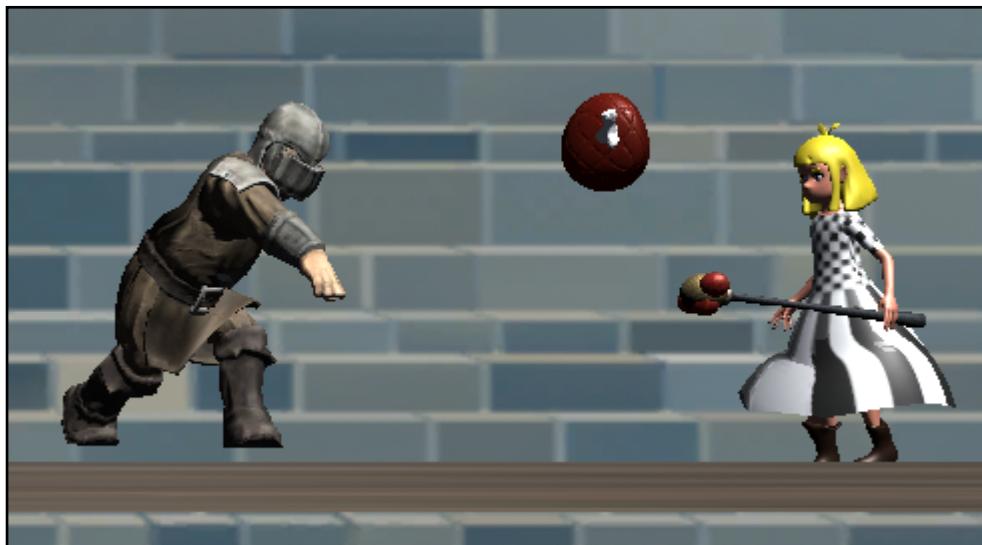
#### Rzut szynką:

Po rozpoczęciu fazy drugiej „Naczelnik” otrzymuje możliwość rzutu bronią, gdy gracz znajdzie się w określonej odległości (listing 15.35.). W ramach omawianej umiejętności „Naczelnik” posiada dwie animacje – rzutu oraz wyjmowania nowej broni. Po wyrzucie, broń trzymana w ręce przeciwnika zostaje dezaktywowana do momentu ustawienia wyzwalacza animacji dobywania broni.

```
if (distance <= attackRange + epsilon)
{
    Attack(animator);
}
else if(distance <= attackRange * 3.5 && distance >= attackRange * 2.5)
{
    if (!alreadyThrewed)
    {
        monoBehaviour.StartCoroutine(ThrowWeaponCoroutine(animator));
    }
}
```

Listing 15.35. Fragment kodu odpowiedzialnego za rzucanie bronią

Funkcjonalność rzutu została zdefiniowana w metodzie „ThrowWeaponCoroutine” typu „IEnumerator”. Jest to szczególny rodzaj interfejsu w języku C#, który umożliwia asynchroniczne wykonywanie operacji tj. opóźnienia. W poniższej metodzie zastosowano opóźnienia przed zmianą widoczności szynki trzymanej w dłoni Naczelnika. Dzięki temu zachowano spójność pomiędzy animacjami, a pojawiającą się bronią (rys. 15.26. przedstawia rzut szynką w gracza).



Rys. 15.26. Animacja ataku szynką

Podczas tworzenia kopii broni Naczelnika, zostaje ona odpowiednio przeskalowana oraz otrzymuje właściwości, które powodują że gracz otrzyma obrażenia, gdy wejdzie w kolizję z rzucaną szynką. Siłę rzutu można dostosować korzystając z publicznych pól „throwForce” oraz „throwUpwardForce”. Na listingu 15.36. przedstawiono metodę „ThrowWeaponCoroutine”.

```

private IEnumerator ThrowWeaponCoroutine(Animator animator)
{
    alreadyThrewed = true;

    animator.SetTrigger("Attack_Throw");

    yield return new WaitForSeconds(1.1f);

    GameObject thrownWeapon = Instantiate(bossWeapon, throwPoint.position, throwPoint.rotation);

    bossWeapon.SetActive(false);

    thrownWeapon.tag = weaponCloneTag;

    thrownWeapon.transform.localScale = new Vector3(1f, 1f, 1f);

    Rigidbody weaponRb = thrownWeapon.GetComponent<Rigidbody>();

    BoxCollider weaponCollider = thrownWeapon.GetComponent<BoxCollider>();

    if (weaponRb == null)
    {
        weaponRb = thrownWeapon.AddComponent<Rigidbody>();
    }

    if (weaponCollider == null)
    {
        weaponCollider = thrownWeapon.AddComponent<BoxCollider>();
    }

    Vector3 throwDirection = (player.position - throwPoint.position).normalized;

    Vector3 forceToAdd = throwDirection * throwForce + Vector3.up * throwUpwardForce;

    weaponRb.AddForce(forceToAdd, ForceMode.Impulse);

    yield return new WaitForSeconds(1.3f);

    bossWeapon.SetActive(true);

    Destroy(thrownWeapon);

    alreadyThrewed = false;
}

```

*Listing 15.36. Metoda ThrowWeaponCoroutine*

## Leczenie:

Wywołanie umiejętności „Leczenia” następuje w metodzie „TakeDamage” (listing 15.37.), gdy poziom zdrowia „Naczelnika” osiągnie 25% maksymalnego zdrowia oraz nie używa on umiejętności „Leczenia”. Podczas aktywnej umiejętności „Leczenia” siada na podłożu i zaczyna regenerować zdrowie (rys.15.27.). Po zakończeniu lub przerwaniu tego procesu, przeciwnik wstaje i kontynuuje atakowanie gracza.

```

1 reference
public void TakeDamage()
{
    boss_1_health -= 1;
    healthBar.value = boss_1_health;
    if (boss_1_health < bosshalfhp)
    {
        if (animator != null)
        {
            animator.SetBool("Is_Enraged", true);
            boss_1_speed = enragesspeed;
        }
    }
    if (boss_1_health <= bossquarterhp && animator.GetBool("Have_Healed") == false )
    {
        animator.SetBool("Is_Healing", true);
    }

    if (boss_1_health <= 0)
    {
        animator.SetTrigger("Is_Dying");
    }
}

```

Listing 15.37. Warunek wywołania umiejętności leczenia



Rys. 15.27. Animacja leczenia naczelnika

Gdy umiejętność „Leczenia” jest aktywowana zmieniona zostaje wartość boolowska „Have\_Healed” na „true” w celu zablokowania ponownego użycia umiejętności, a następnie wartość życia „Naczelnika” zwiększa się wraz z upływem czasu aż osiągnie 50% maksymalnego zdrowia. Wartość zdrowia naczelnika wzrasta 1 punkt/s. Klasę odpowiedzialną za „Leczenie” przedstawia listning 15.38.

```

public class Boss_Healing : StateMachineBehaviour
{
    Boss_Stats bossStats;
    private int halfhp;
    private float healingRate = 1.0f;
    private float elapsedTime = 0.0f;

    0 references
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        bossStats = animator.GetComponent<Boss_Stats>();
        if (bossStats != null)
        {
            halfhp = bossStats.bosshalfhp;
        }
    }

    0 references
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        animator.SetBool("Have_Healed", true);
        if (bossStats.boss_1_health == halfhp) { animator.SetBool("Is_Interrupted", true); }

        elapsedTime += Time.deltaTime;

        if (elapsedTime >= healingRate && bossStats.boss_1_health < halfhp && !animator.GetBool("Is_Interrupted"))
        {
            animator.SetBool("HealingRightNow", true);

            bossStats.boss_1_health++;
            Debug.Log("Current Boss Health: " + bossStats.boss_1_health);
            elapsedTime = 0.0f;
        }
    }
}

```

*Listing 15.38. Kod odpowiedzialny za umiejętnośc Leczenia*

Poniższy listing (15.39.) przedstawia wykorzystanie flagi do sprawdzenia czy „Naczelnik” używa swojej umiejętności w danym momencie. Jeżeli otrzyma on obrażenia i jest w trakcie jej wykonywania, zostanie ona automatycznie przerwana i nie będzie mogła być użyta ponownie.

```

0 references
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Weapon") && playerAnimation.isAttacking)
    {
        TakeDamage();
        playerAnimation.isAttacking = false;
        if (animator.GetBool("HealingRightNow") == true) {
            animator.SetBool("Is_Interrupted", true);
        }
    }
}

```

*Listing 15.39. Wykorzystanie flagi HealingRightNow*

Pokonanie Naczelnika następuje w momencie gdy poziom jego życia osiągnie 0. Wtedy też uruchamiana jest animacja śmierci. Listning 15.40. przedstawia warunek aktywacji danej animacji.

```
if (boss_1_health <= 0)
{
    animator.SetTrigger("Is_Dying");
}
```

Listing 15.40. Kod odpowiedzialny za wywołanie animacji śmierci

Do animacji został dodany „Event” dzięki, któremu po ukończeniu animacji zostanie wywołana metoda „Die”. Metoda ta odpowiada za wyświetlenie efektu zakończenia poziomu (rys. 15.28.) oraz załadowanie sceny wyboru poziomu.



Rys. 15.28. Widok animacji śmierci naczelnika

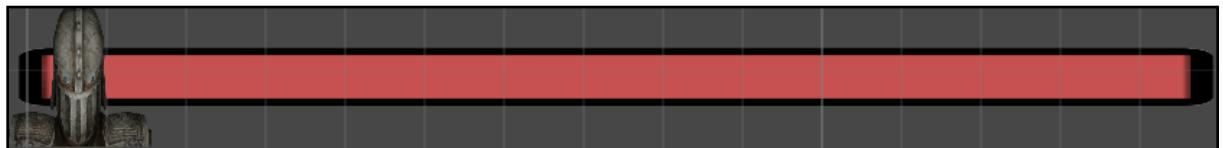
Listing 15.41. przedstawia procedurę „Die”. Pozycja efektu została przypisana aktualnej pozycji gracza. „PlayerPrefs” odpowiada za odblokowanie dostępności następnego poziomu. Scena wyboru poziomu zostanie załadowana po zakończeniu efektu ukończenia poziomu.

```
public void Die()
{
    if (fanfareParticle != null)
    {
        Vector3 spawnPosition = fanfareSpawnPoint != null ? fanfareSpawnPoint.position : transform.position;
        spawnPosition += Vector3.up * 4f;
        ParticleSystem particleInstance = Instantiate(fanfareParticle, spawnPosition, Quaternion.identity);
        particleInstance.Play();
        float particleDuration = 2f;
        PlayerPrefs.SetInt("UnlockedLevel", PlayerPrefs.GetInt("UnlockedLevel", 1) + 1);

        Invoke("LoadLevelScene", particleDuration);
    }
}
```

Listing 15.41. Metoda Die

W celu wizualizacji poziomu zdrowia „Naczelnika” został wykorzystany suwak (Rys. 15.29.), do którego został podłączony i edytowany skrypt „Boss\_Stats”. Na listingach 15.42. oraz 15.43. zostało zaprezentowanie przypisanie wartości poziomu zdrowia do suwaka.



Rys. 15.29. Pasek zdrowia naczelnika

```
public class Boss_Stats : MonoBehaviour
{
    [Header("Stats")]
    public int boss_1_health;
    public int boss_1_speed;
    public int boss_1_attackRange;
    public Slider healthBar;
```

Listing 15.42. Utworzenie zmiennej paska zdrowia

```
public void TakeDamage()
{
    boss_1_health -= 1;
    healthBar.value = boss_1_health;
```

Listing 15.43. Zmiana wartości paska zdrowia

Listing 15.44. odpowiada za inicjalizację zmiennych podczas uruchomienia aplikacji, dzięki czemu wartość maksymalnego zdrowia oraz początkowa wartość zdrowia Naczelnika zostaje ustawiona automatycznie.

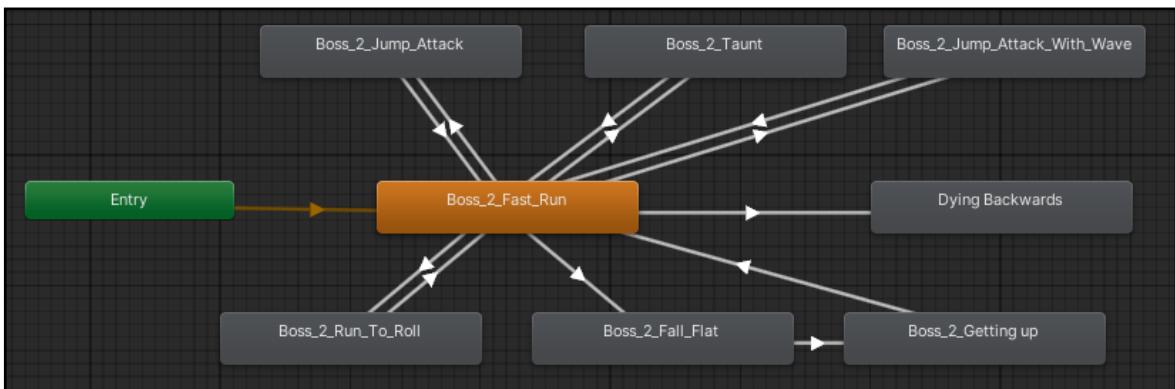
```
private void Start()
{
    playerAnimation = FindObjectOfType<PlayerAnimation>();
    animator = GetComponent<Animator>();

    bosshalfhp = boss_1_health / 2;
    bossquarterhp = boss_1_health / 4;
    enragespeed = boss_1_speed * 2;
    boss_1_max_Hp = boss_1_health;
    healthBar.MaxValue = boss_1_max_Hp;
    healthBar.value = boss_1_health;
}
```

Listing 15.44. Inicjalizacja zmiennych

### 15.3.2. Rozjuszony Naczelnik Więzienia

W celu utworzenia przeciwnika specjalnego „Rozjuszony Naczelnik Więzienia”, wykorzystany został model „Leniwego Naczelnika Więzienia”. Wszystkie animacje oraz kod odpowiedzialny za atakowanie zostały zastąpione nowymi. Ponownie został wykorzystany skrypt do przemieszczania się oraz do obracania obiektu. Usunięta została 2 faza przeciwnika. Na rysunku 15.30. został przedstawiony animator „Rozjuszonego Naczelnika”



Rys. 15.30. Animator Rozjuszonego Naczelnika

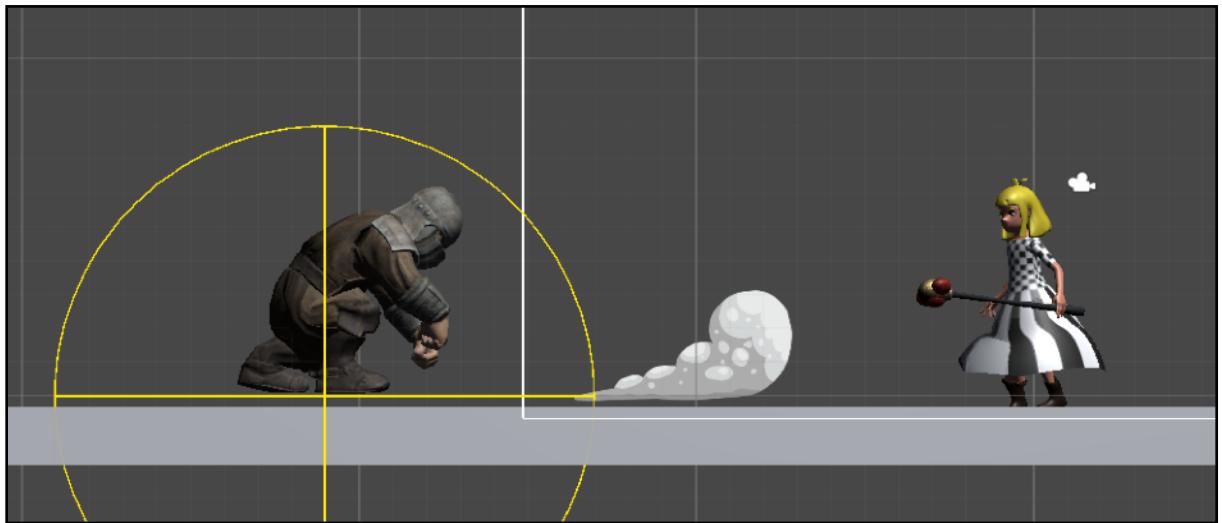
Za uruchomienie animacji odpowiedzialny jest kod podany na listingu nr 15.45. Jest to kod zbliżony do ataku „Białego Piona”. W przypadku „Rozjuszonego Naczelnika” do wywołania „Skoku” wymagane jest również, aby gracz wykonał skok poprzez naciśnięcie odpowiedniego przycisku.

```
private void JumpAttackRunner()
{
    if (!cooldownJump)
    {
        jumpCooldownTime -= Time.deltaTime;
        if (jumpCooldownTime <= 0f)
        {
            cooldownJump = true;
        }
    }

    if (PlayerInKnockBackArea() && cooldownJump && Input.GetKeyDown(jumpKey))
    {
        animator.SetTrigger("JumpAttack");
        cooldownJump = false;
        jumpCooldownTime = initialJumpCooldownTime;
    }
}
```

Listing 15.45. Metoda JumpAttack Runner

Metody odpowiedzialne za uruchamianie innych ataków są do siebie zbliżone i różnią się jedynie wartościami. W przypadku innych metod nie jest wymagane aby gracz wykonał naciśnięcie przycisku. Przez zbieżność w kodzie pozostałe metody odpowiedzialne za atakowanie nie zostaną opisane. Po wykonaniu metod odpowiedzialnych za skok lub przewrócenie tj.: „JumpAttackRunner”, „JumpWithWaveAttackRunner”, „FallFlatAttackRunner” w miejscu uderzenia zostaje utworzona fala uderzeniowa zaprezentowana na rysunku 15.31.



Rysunek 15.31. Animacja uderzenia naczelnika

Za utworzenie fali uderzeniowej poruszającej się w stronę gracza odpowiedzialny jest kod na listingu 15.46. Dzięki pobraniu obecnego czasu trwania animacji możliwe jest wywołanie eksplozji w wybranej klatce animacji. W celu zapobiegnięcia ponownego uruchomienia animacji wybucha, została wykorzystana flaga „haveExploded”.

```

override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    float animationTime = stateInfo.normalizedTime * stateInfo.length;
    if (!haveExploded && animationTime >= 2.07f)
    {
        WywolajWybuch();
        haveExploded = true;
    }
}

1 reference
public void WywolajWybuch()
{
    if (prefab != null && rb != null)
    {
        Vector3 currentPosition = rb.transform.position;
        float newYPosition = currentPosition.y + 1.0f;
        Vector3 newPosition = new Vector3(currentPosition.x, newYPosition, currentPosition.z);
        GameObject prefabInstance1 = Instantiate(prefab, newPosition, Quaternion.identity);
        Rigidbody prefabRigidbody1 = prefabInstance1.GetComponent<Rigidbody>();
    }
}

```

Listing 15.46. Skrypt obsługujący falę uderzeniową

Metoda „WywolajWybuch” pobiera współrzędne obiektu, następnie zostają one zmienione w celu poprawnego umiejscowienia wybucha. Wykorzystując poprawione współrzędne, zostaje utworzony wzorzec wybucha wraz z właściwością „RigidBody”.

Aby dany obiekt uzyskał możliwość poruszania się w stronę gracza, powstał kod zawarty na listingu 15.47. oraz 15.48. Metoda „playerPosition” jest odpowiedzialna za określenie zwrotu wektora prędkości. W zależności od pozycji gracza w stosunku do obiektu, zwracana jest wartość prędkości – ujemna gdy gracz znajduje się po lewej stronie obiektu, w innym przypadku zwrócona jest wartość dodatnia.

```

    float playerPosition()
    {
        if (player != null)
        {
            if (rb.position.x > player.position.x)
            {
                return -inicialspeed;
            }
            else
            {
                return inicialspeed;
            }
        }
        else
        {
            return inicialspeed;
        }
    }
}

```

*Listing 15.47. Metoda playerPosition*

```

override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    speed = playerPosition();
    rb.velocity = new Vector3(speed, 0, 0);

    if (speed < 0 && !haveSwitched)
    {
        transform.rotation = Quaternion.Euler(0, 180, 0);
        haveSwitched = true;
    }
    else if (speed > 0 && !haveSwitched)
    {
        transform.rotation = Quaternion.identity;
        haveSwitched = true;
    }

    float scaleRate = 0.4f;
    scaleFactor += scaleRate * Time.deltaTime;
    scaleFactor = Mathf.Clamp(scaleFactor, 1.0f, 2.0f);
    Vector3 newScale = Vector3.one * scaleFactor;
    transform.localScale = newScale;

    if (scaleFactor >= 2.0f)
    {
        Destroy(animator.gameObject);
    }
}

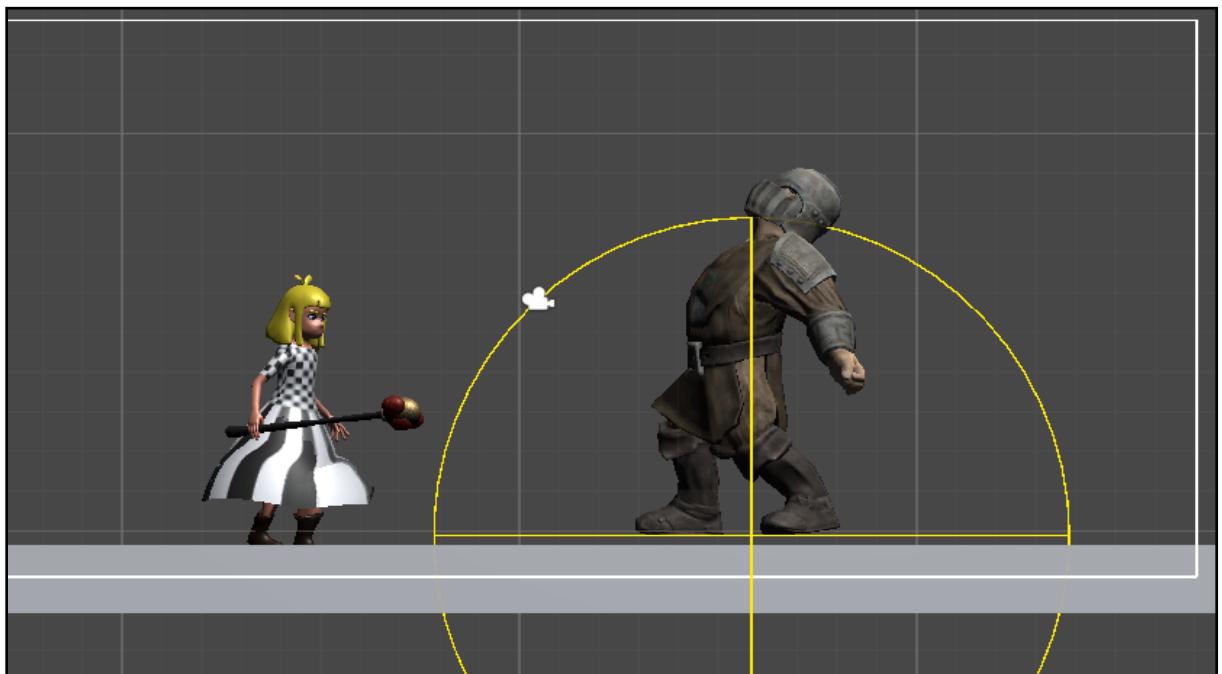
```

*Listing 15.48. Metoda playerPosition*

Zwrot oraz prędkość obiektu nadawane są w metodzie „OnStateUpdate”. W odróżnieniu od metody „Update”, „OnStateUpdate” jest wykorzystywane w przypadku pracy nad animacjami.

Po nadaniu zwrotu oraz prędkości następuje zmiana skali danego obiektu. Dzięki wykorzystaniu funkcji „Clamp” zostaje zdefiniowana maksymalna oraz minimalna wartość skali. Zniszczenie wzorca następuje gdy wartość jego skali wynosi dwukrotność skali początkowej.

Po wywołaniu metody odpowiedzialnej za „Okrzyk bojowy” zostanie wykonana animacja „Taunt”, zaprezentowana na rys. 15.32.



Rys. 15.32. Animacja okrzyku bojowego

UmiejĘtnośĆ „Okrzyk bojowy” jest atakiem, którego celem jest odrzucenie gracza od „Naczelnika”. Za odrzut odpowiedzialny jest kod zawarty na listingu 15.49. W tym celu do obiektu gracza dodana jest siła z zwrotem zależnym od jego pozycji. Odrzut odbywa się jedynie w płaszczyźnie osi X.

```
override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    if (player != null)
    {
        Rigidbody playerRigidbody = player.GetComponent<Rigidbody>();

        if (playerRigidbody != null)
        {
            Vector3 knockbackDirection = player.position - animator.transform.position;
            knockbackDirection.y = 0f;
            knockbackDirection.Normalize();
            float smallerKnockBackForce = knockBackForce * 0.5f;
            playerRigidbody.AddForce(knockbackDirection * smallerKnockBackForce, ForceMode.Impulse);
        }
    }
}
```

Listing 15.49. Skrypt obsługujący okrzyk bojowy

Listing 21

### 15.3.3. Biały Król

Model Białego Króla został zaprezentowany na rys. 15.33. Obiekt utworzono w programie do grafiki trójwymiarowej Blender. Do utworzenia modelu zastosowano technikę rzeźbienia w celu nadania detali.



Rys. 15.33. Model Białego Króla

Zastosowanie techniki rzeźbienia umożliwia rzeźbienie kształtów, jest ona powszechnie stosowana w przypadku modelowania postaci. Narzędzia „Pędzel”, „Złap” czy „Elastyczna Deformacja” umożliwiają zmianę kształtu i detali obiektu. Na rys 15.34. zaprezentowano „Białego Króla” oraz „Ico sfera” będącą obiektem początkowym do wytworzenia głowy „Białego Króla”



Rys. 15.34. Widok detali modelu białego króla

W programie Blender, obiekty złożone są z wielokątów (są to składniki podstawowej powierzchni), które tworzą siatkę reprezentującą strukturę obiektu. Jest ona złożona z wierzchołków, krawędzi oraz wielokątów. Zastosowanie techniki rzeźbienia pozwala na swobodniejszą pracę wraz z siatką obiektu, jednak wymaga ona większego nakładu pracy oraz jest techniką trudniejszą do opanowania dla początkujących. W celu utworzenia obiektu z większą ilością detali, należy zwiększyć ilość wielokątów dzielącą dany obiekt. Zostało to zaprezentowane na rysunku 15.34., gdzie obiekt znajdujący się po lewej stronie rysunku ma domyślną wartość wielokątów dla obiektu „Ico sfera”, natomiast wartość wielokątów dzieląca obiekt znajdujący się po prawej stronie rysunku została zwiększona.

### Umiejętność pasywna:

Spadające obiekty utworzono metodą zaprezentowaną na listingu 15.50. Do obsługi umiejętności pasywnej został wykorzystany kod opierający się o „czas odnowienia”. Kod ten został opisany w kontekście „Białego Piona” (rozdział 15, podpunkt 1.1). W tym przypadku atak został zastąpiony utworzeniem wzorca w jednym z trzech miejsc.

```
1 reference
private void SpawnRockRunner() {
    if (!cooldownSpawnRock)
    {
        SpawnRockCooldownTime -= Time.deltaTime;
        if (SpawnRockCooldownTime <= 0f)
        {
            cooldownSpawnRock = true;
        }
    }

    if (cooldownSpawnRock)
    {
        int chosePoint = Random.Range(1, 3);
        Debug.Log(chosePoint);
        switch (chosePoint) {
            case 1:
                GameObject prefabInstance1 = Instantiate(RockPrefab, PointToRespawnRockA.transform.position, Quaternion.identity);
                break;
            case 2:
                GameObject prefabInstance2 = Instantiate(RockPrefab, PointToRespawnRockB.transform.position, Quaternion.identity);
                break;
            default:
                GameObject prefabInstance3 = Instantiate(RockPrefab, PointToRespawnRockC.transform.position, Quaternion.identity);
                break;
        }

        cooldownSpawnRock = false;
        SpawnRockCooldownTime = initialRockCooldownTime;
    }
}
```

*Listing 15.50. Metoda SpanRockRunner*

W celu zniszczenia obiektu kamienia powstał kod znajdujący się na listingu 15.51. Podobny kod został zastosowany do obsługi poruszania się „Białego Skoczka” (rozdział 15, podpunkt 1.2.). Jeżeli dany obiekt wszedł w kolizję z podłożem, graczem lub przeciwnikiem następuje zniszczenie obiektu oraz wywołanie animacji.

```

0 references
void Update()
{
    Collider[] enemyCollidersUsingTagPointA = Physics.OverlapSphere(groundChecker.transform.position, sphereRadius);
    touchedGround = enemyCollidersUsingTagPointA.Any(collider => collider.CompareTag("Ground"));
    touchedEnemy = enemyCollidersUsingTagPointA.Any(collider => collider.CompareTag("Enemy"));
    touchedPlayer = enemyCollidersUsingTagPointA.Any(collider => collider.CompareTag("Player"));
    if (touchedEnemy || touchedGround || touchedPlayer) { if (!isTriggered) { DestroyEnemy(); } }
}
1 reference
private void DestroyEnemy()
{
    isTriggered = true;
    Vector3 enemyPosition = transform.position;

    StartCoroutine(WaitForDeathAnimation(enemyPosition));
}
1 reference
private IEnumerator WaitForDeathAnimation(Vector3 enemyPosition)
{
    Instantiate(explosionEffectPrefab, enemyPosition, Quaternion.identity);
    yield return new WaitForSeconds(0.2f);
    Destroy(gameObject);
}

```

*Listing 15.51. Fragment służący do usuwania kamieni*

### Poruszanie się:

Za poruszanie się oraz zwrot „Białego Króla” odpowiada wcześniej już zastosowany kod służący do obsługi poruszania się „Leniwego Naczelnika Więzienia” (rozdział 15, podpunkt 3.1.).

### Umiejętności specjalne:

W przypadku umiejętności „tarczy” (Listing 15.52.) utworzony zostaje obiekt reprezentujący tarczę o odpowiednich współrzędnych. W celu zaimplementowania „tarczy” zmodyfikowany został fragment kodu „Take Damage” zawarty na listingu 15.53. Wykorzystując flagę „isShielded” zostaje zablokowane otrzymywanie obrażeń przez „Białego Króla”, jeżeli wartość zdrowia tarczy spadnie do 0, jej obiekt zostaje zniszczony, a wartość flagi „isShielded” jest zmieniana na „false”. Implikuje to ponowną możliwość otrzymywania obrażeń przez „Białego Króla”.

```

private void SpawnShieldRunner()
{
    if (!isShielded)
    {
        if (!cooldownSpawnShield)
        {
            SpawnShiledCooldownTime -= Time.deltaTime;
            if (SpawnShiledCooldownTime <= 0f)
            {
                cooldownSpawnShield = true;
            }
        }

        if (cooldownSpawnShield)
        {
            shieldHp = maxshieldHp;
            ShieldInstance = Instantiate(ShieldPrefab, PointToRespawnShields.transform.position, Quaternion.identity);
            ShieldInstance.transform.parent = rb.transform;
            cooldownSpawnShield = false;
            isShielded = true;
            SpawnShiledCooldownTime = initialShieldCooldownTime;
        }
    }
}

```

*Listing 15.52. Metoda SpawnShieldRunner*

```

public void TakeDamage()
{
    if (isShielded) { shieldHp--; if (shieldHp <= 0) { Destroy(ShieldInstance); isShielded = false; } }
    else if (!isShielded)
    {
        hp--;
        if (hp <= 0) Invoke(nameof(DestroyEnemy), 0.5f);
    }
}

```

*Listing 15.53. Metoda TakeDamage*

Za umiejętność specjalną przyzwaną „Białej Wieży” odpowiedzialny jest kod zawarty na listingu 15.54. Tu również zastosowano podejście oparte na „czasie odnowienia”, jednak w tym przypadku sprawdzane jest również czy na danym punkcie do przyzwania nie znajduje się już obiekt. Jeżeli oba punkty są zajęte przez „Białe Wieże” wtedy też zostaje zatrzymane odliczanie czasu do kolejnego przyzwania.

```

private void SpawnRookRunner()
{
    if (!pointAOccupied || !pointBOccupied)
    {
        if (!cooldownSpawnRook)
        {
            SpawnRooktCooldownTime -= Time.deltaTime;
            if (SpawnRooktCooldownTime <= 0f)
            {
                cooldownSpawnRook = true;
            }
        }

        if (cooldownSpawnRook)
        {
            if (!pointBOccupied)
            {
                GameObject prefabInstance1 = Instantiate(RookPrefab,
                    PointToRespawnTowerB.transform.position, Quaternion.Euler(0f, -90f, 0f));
            }
            if (!pointAOccupied && pointBOccupied)
            {
                GameObject prefabInstance1 = Instantiate(RookPrefab,
                    PointToRespawnTowerA.transform.position, Quaternion.Euler(0f, -90f, 0f));
            }
            cooldownSpawnRook = false;
            SpawnRooktCooldownTime = initialRookCooldownTime;
        }
    }
}

```

*Listing 15.54. Metoda SpawnRookRunner*

W podobny sposób zostaje przywoływany „Biały Goniec”.

#### 15.3.4. Czarny Król

Model „Czarnego Króla” znajdujący się na rys. 15.35 jest to wcześniej zaprezentowany model „Białego Króla” zaprezentowane w poprzednim podpunkcie. W celu rozróżnienia modeli zmienione zostały kolory oraz broń.



Rys. 15.35. Model Czarnego Króla

#### Poruszanie się:

„Czarny Król” tak jak i „Biały Król” przemieszcza się dzięki skryptowi, który odpowiada za poruszanie się „Naczelnika”

#### Umiejętności specjalne:

Umiejętność przyzywania jest metodą identyczną jak w przypadku „Białego Króla”, jedyną zmianą jest wzorzec wybranych przeciwników. Białe figury zostały zastąpione czarnymi.

Metoda odpowiedzialna za utworzenie kolców jest zbliżona do metody zaprezentowanej w przypadku utworzenia efektu wywoływanego podczas skoku „Rozjuszonego Naczelnika Więzienia” (rozdział 15, podpunkt 3.1.).

W przypadku umiejętności obsługującej piorun (listing 15.55.), ponownie zastosowano „czas odnowienia” oraz oczekiwanie na skok gracza.

```
private void ThunderBoltRunner() {
    if (!cooldownSpawnThunder)
    {
        SpawnThunderCooldownTime -= Time.deltaTime;
        if (SpawnThunderCooldownTime <= 0f)
        {
            cooldownSpawnThunder = true;
        }
    }

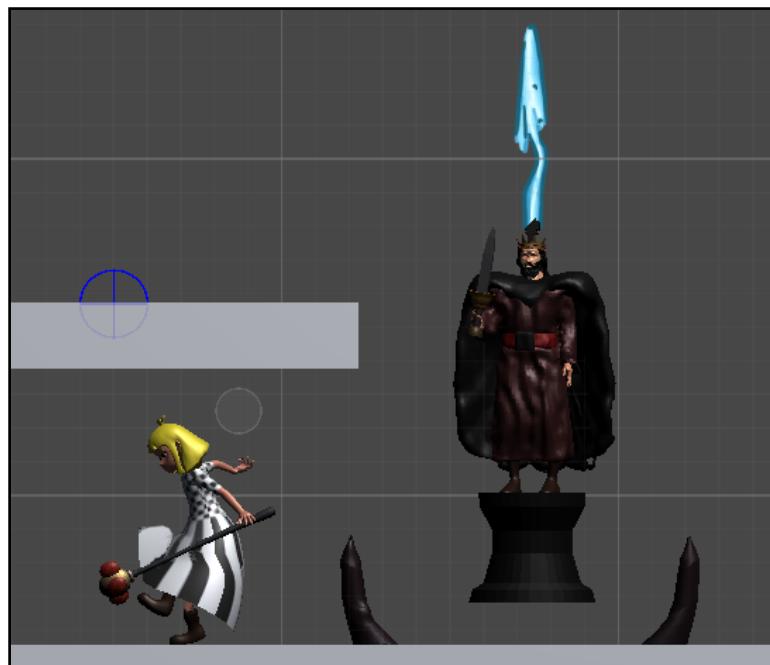
    if (cooldownSpawnThunder && Input.GetKeyDown(jumpKey))
    {
        GameObject prefabInstance1 = Instantiate(ThunderBoltPrefab,
            PointForThunderBolt.transform.position, Quaternion.Euler(0f, 0, -180f));

        prefabInstance1.transform.parent = transform;
        cooldownSpawnThunder = false;
        SpawnThunderCooldownTime = initialThunderCooldownTime;

        Destroy(prefabInstance1, 3f);
    }
}
```

Listing 15.55. Metoda ThunderBoltRunner

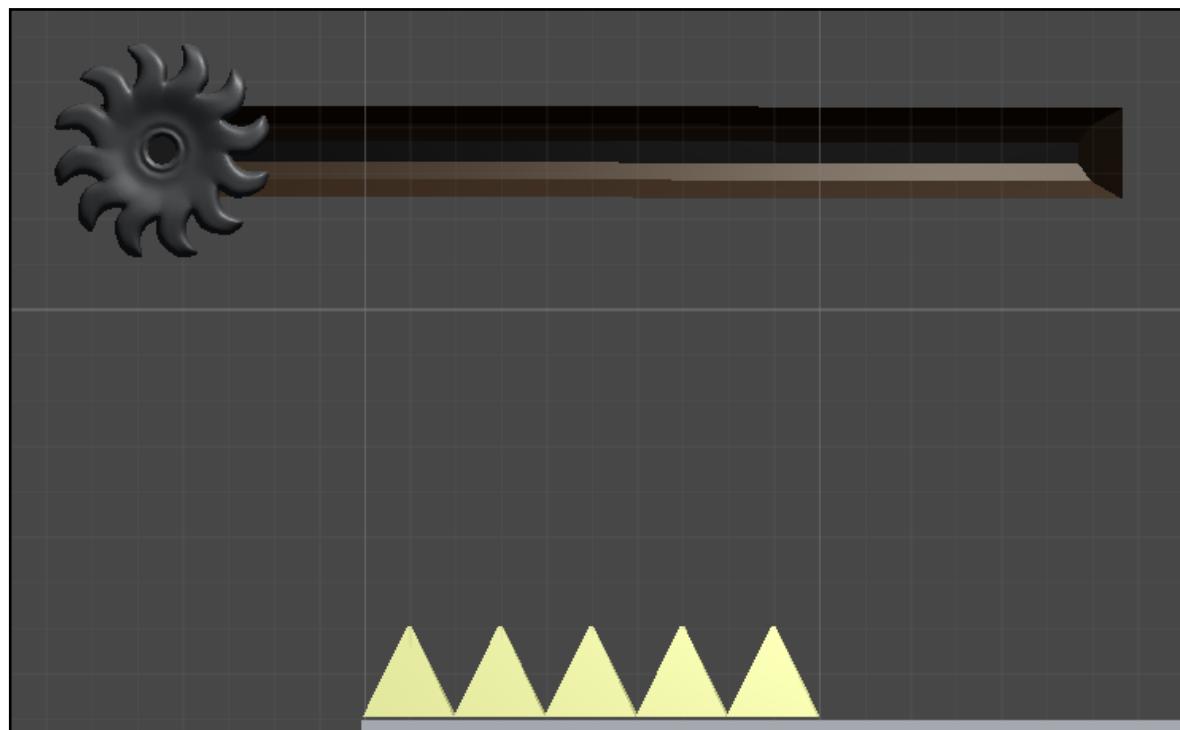
Po wywołaniu metody piorun zostaje zniszczony po określonym czasie. W celu poprawnego zadawania obrażeń zmieniono rozmiar „Box Collidera” w zależności od wielkości animacji. Rys. 15.36. przedstawia umiejętności „Czarnego Króla”



Rys. 15.36. Efekt umiejętności specjalnej czarnego króla

## 15.4. Pułapki

Obiekty „Piły Tartaku” oraz „Kolców” zaprezentowane zostały na rysunku 15.37. Zadawanie obrażeń zostało obsłużone w identyczny sposób w jaki robią to przeciwnicy (rozdział 13, podpunkt 5), a za ruch „Piły” odpowiedzialny jest kod służący do przemieszczania się „Białego Piona” (rozdział 15, podpunkt 1.1.).



Rys. 15.37. Modele pułapek.

## 16. Implementacja Widoków Aplikacji

Aplikacje tworzone w środowisku Unity opierają się na systemie scen. Z każdą sceną użytkownik może powiązać zestaw zasobów („assetów”) oraz skrypty. W początkowej fazie implementacji projektu utworzono sceny prezentujące menu gry(menu główne, ustawienia, wybór poziomu). Tła do tych scen zostały wyrenderowane z utworzonych modeli w programie Blender.

### 16.1. Scena Menu głównego

Rysunek 16.1. przedstawia menu główne gry, w którym gracz może rozpoczęć lub wczytać grę, dostosować ustawienia oraz wyjść z gry. Przyciski służące do nawigacji są elementem pakietu „TextMeshPro” dostępnego w Unity. Do przycisków zostały przypisane odpowiednie metody skryptu „Menu” (rozdział 13, podpunkt 6.).



Rys 16.1. Scena menu głównego

### 16.2. Ekran wyboru profilu gracza

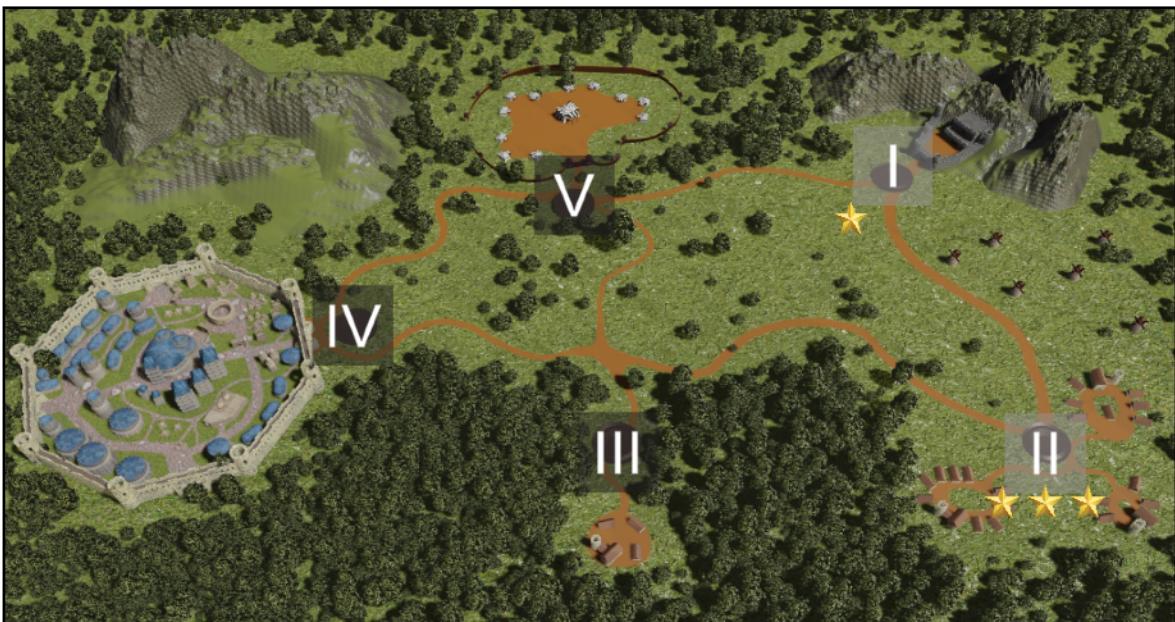
Ekran wyboru profilu gracza (rys. 16.2.) został zawarty w scenie menu głównego. Po naciśnięciu przycisku „Nowa gra” lub „Wczytaj grę”, panel z przyciskami z menu głównego zostanie dezaktywowany. Takie podejście pozwala na ograniczenie liczby scen. Jeżeli na wybranym profilu nie ma zapisanych żadnych danych – na przycisku tego profilu wyświetli się napis „Profil jest pusty”. Przycisk ten będzie nieaktywny po wciśnięciu „Wczytaj grę”, ponieważ na wybranym profilu nie wystąpił żaden zapis. Na przyciskach wyboru profilu znajdują się informacje o sumie zebranych punktów oraz liczbie śmierci gracza. Dane te są zapisywane dynamicznie do pliku JSON podczas rozgrywki, a na ekranie wyboru profilu gracza są one odczytane z tego pliku. Takie podejście umożliwia skrypty do przechowywania danych gry (rozdział 13, podpunkt 4.).



Rys. 16.2. Ekran wyboru profilu

### 16.3. Scena wyboru poziomu

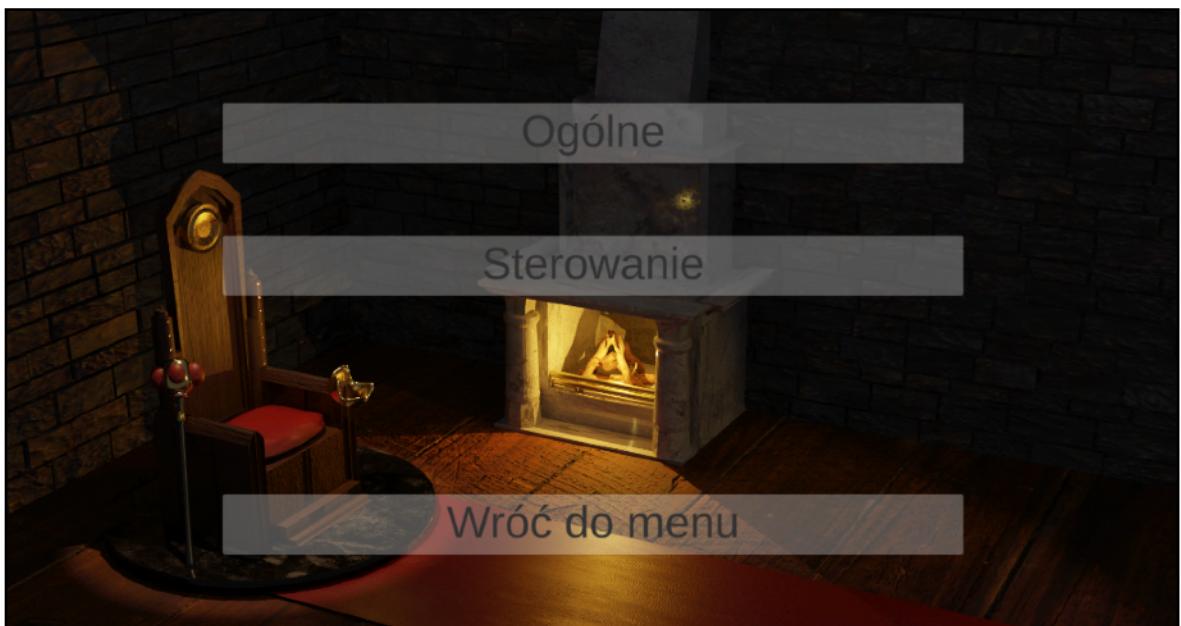
Po wybraniu profilu, gracz zostanie przekierowany na scenę wyboru poziomu, na której znajduje się pięć przycisków, służących do załadowania poziomu o numerze przedstawionym na przycisku (rys. 16.3.). Jeżeli gracz rozpoczął nową grę, jedynym aktywnym przyciskiem będzie poziom I, pozostałe przyciski zostają aktywowane w momencie ukończenia poprzedniego poziomu (np. po pokonaniu bossa). Zablokowane poziomy wyróżniają się czarnym tłem przycisku. Na scenie wyświetla się obecna liczba zebranych punktów na danych poziomach. Sposób działania tej funkcjonalności przedstawiają skrypty odpowiedzialne za system punktacji (rozdział 13, podpunkt 7.). W celu powrotu do sceny menu głównego należy nacisnąć przycisk „ESC”.



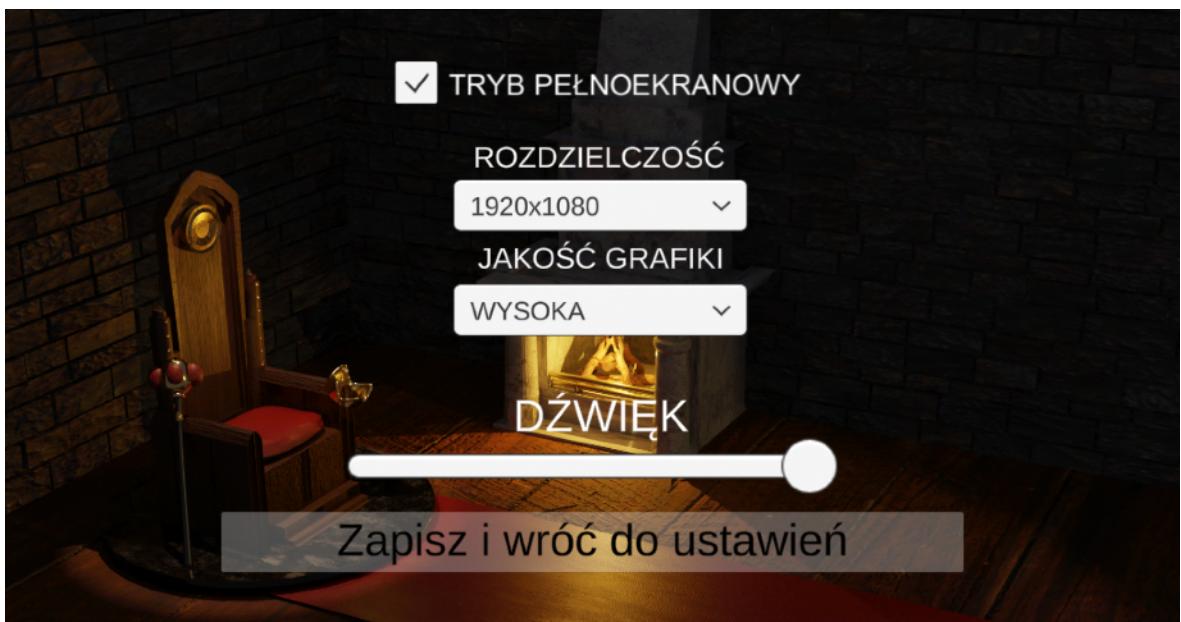
Rys. 16.3. Scena wyboru poziomu

## 16.4. Scena Ustawień

Ustawienia gry zostały podzielone na dwie kategorie – ustawienia „Ogólne” oraz „Sterowanie”. Tło ekranu ustawień jest takie samo jak w menu głównym, jednakże jest to osobna scena. Zawiera ona trzy panele, które są odpowiednio aktywowane lub dezaktywowane. Rysunek 16.4. przedstawia panel wyboru ustawień.



Rys. 16.4. Panel wyboru ustawień



Rys. 16.5. Panel ustawień ogólnych

W panelu ustawień ogólnych (rys. 16.5.) gracz ma możliwość przełączania trybu wyświetlania gry między pełnoekranowym, a okienkowym. Listy rozwijane umożliwiają dostosowanie rozdzielczości oraz jakości grafiki. Suwak daje możliwość zmiany poziomu głośności muzyki w grze, gdyż tak jak pozostałe z wymienionych elementów ma przypisaną odpowiednią metodę klasy „SettingsMenu” (rozdział 13, podpunkt 6.).

W ustawieniach sterowania (rys. 16.6.), gracz ma możliwość sprawdzenia domyślnych klawiszy przypisanych do odpowiadających im akcji gracza oraz może zmienić każdy z tych klawiszy. Po naciśnięciu wybranego przycisku zostanie wyświetlony napis „Wciśnij klawisz”, wtedy gracz może wybrać nowy klawisz przez jego wcisnięcie na klawiaturze lub myszy. Dla każdej z akcji został utworzony pusty obiekt na scenie przechowujący tekst wyświetlany na danym przycisku. Każdy z tych obiektów korzysta ze skryptu "Keybind" (rozdział 13, podpunkt 6.), który umożliwia poprawne działanie systemu przypisania klawiszy do akcji gracza.

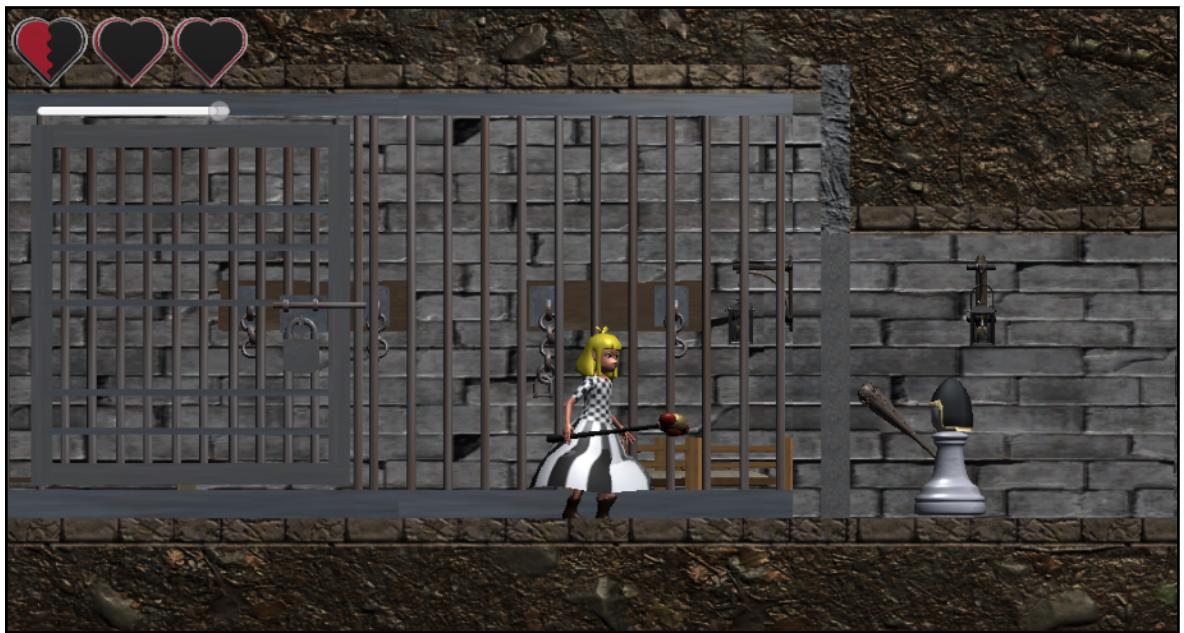


Rys. 16.6. Panel ustawień sterowania

## 16.5. Scena przykładowego poziomu

Podczas rozgrywki, na ekranie gracza wyświetla się obecna liczba życia (na podstawie przeznaczonych do tego skryptów, patrz rozdział 13, podpunkt 6). Jest ona przedstawiona za pomocą ilustracji serc, widocznych na rys. 16.7. Kolejnym elementem widocznym na ekranie jest suwak odmierzający czas do strzału gracza.

Podczas implementacji gry uwzględniono menu pauzy. Jeżeli gracz naciśnie klawisz „ESC” znajdując się na scenie dowolnego poziomu, wtedy gra zostanie wstrzymana oraz wyświetli się panel menu, przedstawiony na rys. 16.8. Funkcjonalności menu pauzy zostały opisane w ramach skryptu „PauseMenu” (rozdział 13, podpunkt 6).

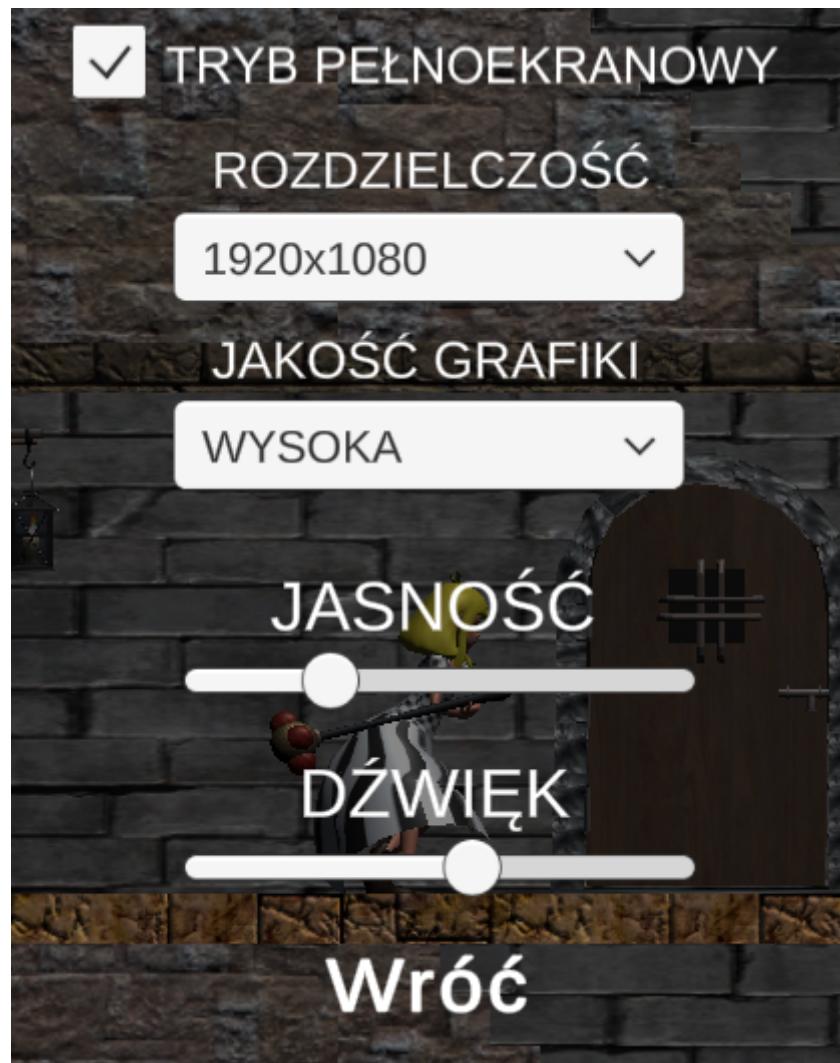


Rys. 16.7. Przykładowy poziom



Rys. 16.8. Menu Pauzy

Ustawienia w menu pauzy wyglądają analogicznie do ustawień ogólnych oraz mają identyczny sposób działania. Jednakże w ustawieniach pauzy znajduje się dodatkowy suwak, dzięki któremu gracz może dostosować jasność podczas rozgrywki (rys. 16.9.). Funkcjonalność zmiany poziomu jasności została opisana w ramach skryptu „Brightness” (rozdział 13, podpunkt 6).



Rys. 16.9. Menu ustawień podczas zapauzowania rozgrywki.

## 17 Testy

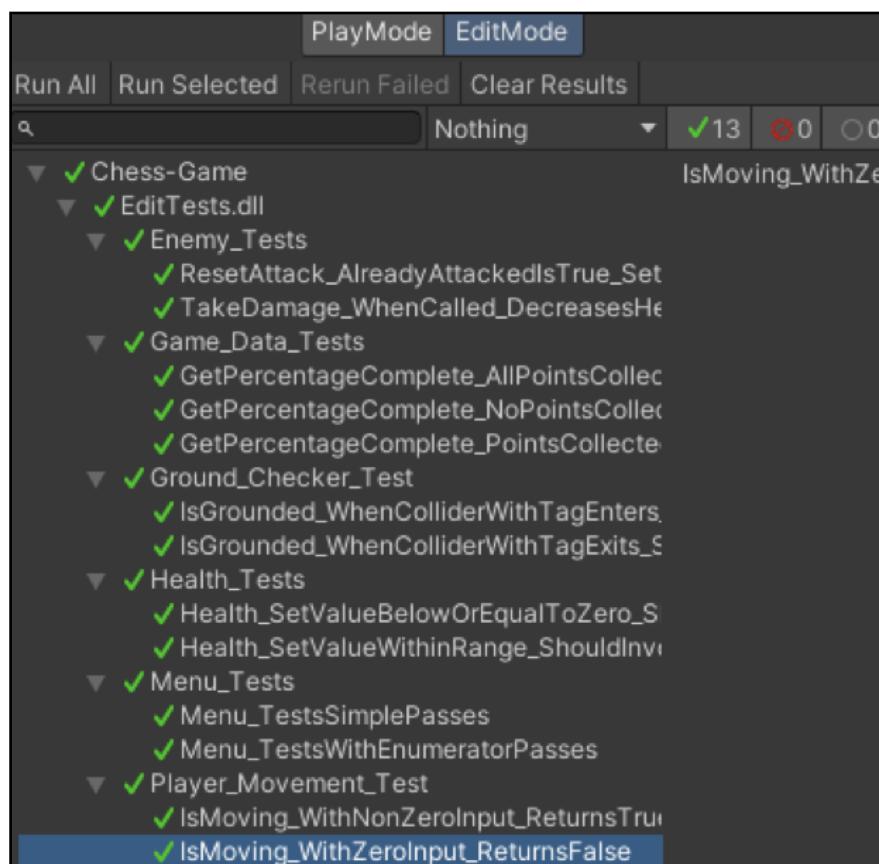
Testowanie aplikacji jest bardzo ważnym punktem każdego projektu. W procesie tworzenia aplikacji dąży się do jak największego pokrycia kodu testami. Idealnym scenariuszem jest pokrycie kodu przynajmniej w 90% testami.

Unity dzieli testy na dwa rodzaje:

- Edit Tests, określane również jako Edit Mode Tests, są to testy przeprowadzane w trybie edycji w środowisku Unity. Testy te skupiają się na sprawdzeniu kodu bez uruchomienia gry. Brak wymogu uruchomienia gry sprawia że są szybsze oraz bardziej efektywne.
- Play Tests, zwane także Play Mode Tests, to testy jednostkowe, które są wykonywane w trakcie działania gry w trybie Play w środowisku Unity. Te testy pozwalają na interakcję z elementami gry w czasie rzeczywistym. Uruchamiają całą grę i sprawdzają, czy różne elementy współpracują poprawnie w dynamicznym środowisku. Testy te są stosowane do testowania funkcji, które wymagają pełnej integracji z silnikiem gry i interakcji z graczem.

Korzystając z obu rodzajów testów, twórcy gier mogą sprawdzać poprawność funkcjonalności swojej aplikacji, zarówno w kontekście izolowanym (Edit Mode), jak również w dynamicznym środowisku gry (Play Tests)

W celu pokrycia kodu powstały testy jednostkowe. Przykładowa lista testów znajduje się na Rys. 17.1.



Rys. 17.1. Lista testów w środowisku Unity

## **18 Wnioski**

W ramach projektu została zaprojektowana oraz zaimplementowana gra platformowa w środowisku Unity. Ma ona na celu promocje warcabów wśród młodzieży. Wykorzystując kolorową oprawę graficzną gra staje się bardziej przystępna dla odbiorcy. W celu zaprojektowania gry zostały zdefiniowane wymagania funkcjonalne oraz niefunkcjonalne obejmujące rozmaite aspekty projektu. Spełnienie ich zapewnia przejrzysty interfejs użytkownika, intuicyjne sterowanie i budowę poziomów, a także dostosowanie gry do sprzętu użytkownika. Projektując sceny aplikacji zastosowano wcześniej zaprojektowane widoki aplikacji w celu zgodności z dokumentacją.

Proces implementacji projektu był zawiły, a w jego trakcie wystąpiło wiele komplikacji. Jednym z problematycznych zagadnień był dostęp do danych między scenami, ponieważ Unity umożliwia dostęp jedynie do aktywnej sceny. Aby rozwiązać te trudność zastosowano lokalny zapis ustawień. Wraz z rozwojem projektu dodawane były różne rozwiązania, jest to widoczne w przypadku poruszania się przeciwników, gdzie początkowo pion poruszał się od punktu „A” do punktu „B”. Natomiast kolejni przeciwnicy zmieniali sposoby swojego poruszania się.

Obecny stan projektu jest zadowalający, jednak pozostała możliwość dalszego rozwoju gry np. dodanie przerywników fabularnych, misji pobocznych lub postaci niezależnych.

Jest wiele składowych zadowalającej produkcji np. udźwiękowienie, grafika, rozgrywka, fabuła. Jednak w przypadku gdy jedna z omawianych składowych jest niskiej jakości może przyćmić jakość innych. Powoduje to, że gry są swego rodzaju sztuką, ponieważ każdy ich aspekt musi być dopracowany, co wymaga czasu oraz wykwalifikowanych ludzi.

## Bibliografia

- (1) 100 mln użytkowników na Chess.com! - <https://www.chess.com/pl/article/view/100-mln-uzytkownikow-na-chess-com> [dostęp 19.01.2024]
- (2) Minkkinen, T. (2016). Basics of Platform Games.
- (3) The Game Awards 2017 rewind <https://thegameawards.com/rewind/year-2017> [dostęp 19.01.2024]
- (4) A Guide to Game Design Process Best Practices (And Blunders) - <https://www.mobileapps.com/blog/game-design-process> [dostęp 19.10.2023]
- (5) Dokumentacja silnika Unity klasy „PlayerPrefs” - <https://docs.unity3d.com/ScriptReference/PlayerPrefs.html> [dostęp 19.01.2024]
- (6) How to design levels for a platformer. - <http://devmag.org.za/2011/07/04/how-to-design-levels-for-a-platformer/> [dostęp 19.10.2023]
- (7) Dokumentacja silnika Unity - <https://docs.unity3d.com/Manual/index.html> [dostęp 19.10.2023]
- (8) AssetsStore silnika Unity - <https://docs.unity3d.com/Manual/AssetStore.html> [dostęp 19.10.2023]
- (9) Dokumentacja języka C# - <https://learn.microsoft.com/en-us/dotnet/csharp/>
- (10) Cooper, James W. "C# design patterns: a tutorial." (2002).
- (11) Lampel, J. (2015). The beginners guide to Blender. Blenderhd. com.
- (12) How to add a texture in Blender. - <https://all3dp.com/2/blender-how-to-add-a-texture/> [dostęp 19.10.2023]
- (13) Lazor, M., Gajić, D. B., Dragan, D., & Duta, A. (2019). Automation of the avatar animation process in FBX file format. *FME Transactions*, 47(2), 398-403.
- (14) Creating animations with Mixamo. - <https://garagefarm.net/blog/creating-mixamo-animations-with-blender> [dostęp 19.10.2023]
- (15) Jacobson, Lvar, and James Rumbaugh Grady Booch. "The unified modeling language reference manual." (2021).
- (16) Dokumentacja narzędzia PlasticSCM - <https://docs.plasticscm.com/zh-cn/> [dostęp 19.01.2024]