UMCS

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
W LUBLINIE
Wydział Matematyki, Fizyki i Informatyki

Kierunek: Informatyka

Dawid Nadzieja
nr albumu: 303842

Wykorzystanie metod logiki rozmytej do sterowania zachowaniem postaci podczas walki w grze typu RPG

The use of fuzzy logic methods to control the character's behavior during combat in an RPG game

Praca licencjacka
napisana w Instytucie Informatyki
pod kierunkiem dr Anny Gajos-Balińskiej

Lublin rok 2023

Spis treści

Wstęp

Gry wideo na dobre zadomowiły się w dzisiejszym świecie i służą nam nie tylko do zabawy, ale również są wykorzystywane w wielu różnych dziedzinach nauk. Warto wspomnieć chociażby o aspektach socjologicznych, psychologicznych czy kulturowych. Oprócz bycia częścią dzisiejszego świata, gry wideo są również piaskownicą, w której można z powodzeniem testować i wdrażać różne rozwiązania technologiczne. Jednym z takich rozwiązań jest wykorzystanie sztucznej inteligencji.

Sztuczna inteligencja odpowiada na problem ciągłej wielowarstwowości gier wideo. Czasy Tetrisa czy Mario odchodzą powoli w zapomnienie, a coraz to nowsze produkcje zaskakują złożonością przedstawionego świata, nieliniowością fabuły czy technologiami żywo imitującymi nasz świat rzeczywisty. Sztuczna inteligencja wykorzystywana jest obecnie w wielu aspektach gier wideo, takich jak symulowanie bohaterów niezależnych, reakcja otoczenia na zachowanie gracza, generacja poziomów czy chociażby w detalach takich jak pojedyncze animacje postaci.

Celem niniejszej pracy licencjackiej jest przedstawienie projektu pewnego elementu gry RPG *(ang. Role-Playing Game)* jakim jest system walki i pokazanie w nim, jak sztuczna inteligencja może wpływać na zachowanie przeciwnika w zależności od tego, co się dzieje na ekranie. W rozdziale pierwszym zostanie omówione dokładnie czym gry RPG są, w jaki sposób można zrealizować sztuczną inteligencję w tego typu grach oraz zostaną scharakteryzowane dostępne technologie pozwalające na wykonanie takiego projektu. W rozdziale drugim zostaną przedstawione główne założenia gry, w tym projekt sztucznej inteligencji przeciwnika. W rozdziale trzecim zostanie zawarta główna część implementacyjna gry, a w rozdziale czwartym omówiona zostanie sama aplikacja oraz zachowanie przeciwnika sterowanego sztuczną inteligencją.

6:1175315206

Rozdział 1

Tworzenie gier RPG

1.1. Historia gier RPG

Gry RPG *(ang. Role-Playing Game)* to bardzo szerokie pojęcie. Jak sama nazwa wskazuje, jest to gra fabularna, w której gracze wcielają się w postać, bądź wiele postaci i odgrywają pewną rolę w fikcyjnym świecie. Pod tym pojęciem można zaklasyfikować mnóstwo różnych gatunków, typów gier, sposobów grania. Do wszelkich odmian można zaliczyć takie rodzaje gier jak gry PnP *(ang. Pen and Paper)*, CRPG *(ang. Computer Role-Playing Game)*, czy chociażby LARP *(ang. Live Action Role Playing)*. Można również wskazać na bardzo popularny podgatunek komputerowych gier, wykorzystujący internet do wspólnej rozgrywki pomiędzy graczami – MMORPG *(ang. Massively Multiplayer Online Role-Playing Game)* [1].

Mnogość gatunków, podgatunków, rodzajów gier, które zbierają się na wspólnotę gier RPG nie zmienia faktu, że większość z nich charakteryzuje się wspólnymi cechami, dzięki którymi możemy zidentyfikować lub zaklasyfikować daną grę jako grę fabularną. Wcześniej wspomniane wcielanie się w postać i odgrywanie ról jest kluczową cechą gier RPG, jednak warto wspomnieć o innych aspektach. Istotnym czynnikiem na pewno jest pewien scenariusz i narracja *(ang. storytelling)*, w której uczestniczą postacie stworzone i sterowane przez graczy, a względem których odbywają się wydarzenia zgodne z zasadami wcześniej przygotowanego, fikcyjnego świata. Gracze przemierzają świat i stawiają czoła różnym wyzwaniom, najczęściej stworzonym i prowadzonym przez mistrza gry *(ang. game master)*. W grach komputerowych rolę mistrza gry zazwyczaj przejmuje silnik gry lub system stworzony wcześniej przez projektantów gry [1]. Różne sytuacje napotkane przez graczy mogą być rozstrzygnięte za pomocą elementu losowości. W grach typu PnP służą do tego kości do gry, w grach komputerowych służy do tego program generatora liczb pseudolosowych.

Historia gier RPG sięga wczesnych lat 70. XX wieku. Powstawały wtedy pierwsze gry planszowe, gry PnP oparte na narracji i tworzeniu postaci. Za prekursora tego gatunku uważa się grę *Dungeons & Dragons* z 1974 roku, która w różnych formach i wersjach jest wydawana po dziś dzień. Tak szybko jak powstały gry planszowe,

gry „tabletop”, tak i również w późniejszych latach 70. powstawały „LARPy” i gry CRPG oparte w dużej mierze na grach PnP i pod szerokim wpływem *Dungeons & Dragons*. Gry komputerowe początkowo były grami tekstowymi, wyświetlanymi w konsoli. Sukcesy takich tytułów jak *Ultima* (1981) czy *Wizardry* (1981) będące luźnymi adaptacjami *Dungeons & Dragons* bardzo mocno spopularyzowały gry RPG w kulturze masowej [2]. Późniejsze wersje tych gier, wraz ze słynnym tytułem *Might & Magic* (1986) markują tzw. „złotą erę” w dziedzinie CRPGów [3]. Wraz z rozwojem technologii, w późniejszych latach 90. powstawały tzw. izometryczne, *sprite-based* gry 2D, takie jak słynne *Diablo* (1997), przedstawione na Rysunku 1.1, *Baldur's Gate* (1998) czy *Icewind Dale* (2000).

Rysunek 1.1: Screen z gry *Diablo* (1997)

Razem z nowym milenium silniki gier 3D zaczęły być masowo wykorzystywane i zdominowały rynek gier komputerowych. Można przytoczyć takie tytuły jak *TES III: Morrowind* (2002), *TES IV: Oblivion* (2004) czy *Fallout 3* (2008) [4]. W tym okresie również do głosu zaczęły dochodzić gry z gatunku MMORPG ze świetnym *World of Warcraft* (2004), który na blisko dwie dekady zacementował swoją pozycję jako najpopularniejsza gra MMORPG.

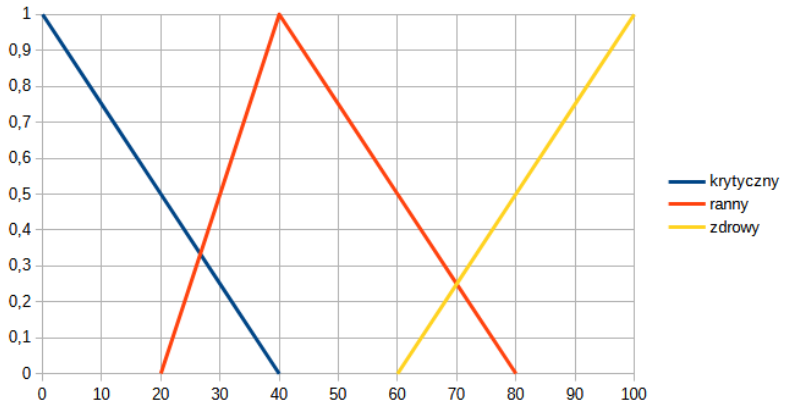
1.2. Sztuczna inteligencja w grach

We wczesnych produkcjach, grach tekstowych, grach ograniczonych złożonością obliczeniową tamtejszego sprzętu, pojawiały się pewne zalążki sztucznej inteligencji. Było to jednak rozwiązanie na kształt „wirtualnego przeciwnika". Proste algorytmy, operujące tylko na kilku twardo zakodowanych regułach i reagujące na podstawowe zachowania gracza (rakietka w *Pongu*, podążanie duszków za graczem w *Pacmanie*) na długi czas dominowały w grach komputerowych [5]. Sytuacja zaczęła się zmieniać wraz z rozwojem technologicznym, dzięki któremu mogły powstać bardziej złożone algorytmy, przetwarzające większą ilość danych. Rozwój sztucznej inteligencji skupił się bardziej ku symulowaniu zachowań ludzkich i zewnętrznego świata. Dziś w grach możemy napotkać takie złożone elementy jak symulacje życia wirtualnego miasta czy drużyn, zespołów bądź przeciwników w grach kompetytywnych, które są w całości obsługiwane przez sztuczną inteligencję.

Sztuczna inteligencja w grach polega w głównej mierze na tworzeniu specyficznych algorytmów w ramach danego modelu. Algorytmy te rozwiązują pewne problemy opierając się na zasadach gry w ramach rozwiązań dostępnych dla gracza. Często w najprostszej postaci są to akcje w stylu „bodziec-reakcja". W bardziej zaawansowanym modelu sztuczna inteligencja na podstawie danych wejściowych musi przeprowadzić pewne wnioskowanie i zdecydować się na jedną z wielu zaprogramowanych możliwości. Problemy rozwiązywane przez AI są bardzo często cząstkowe i rozwiązywane w chwili napotkania przez gracza. Natomiast zbiór takich modeli podczas kreowania i symulacji świata przedstawionego w grze, może już regulować większy element rozgrywki, zwiększając immersję i dając graczowi wrażenie uczestniczenia w „żywym świecie".

Jednymi ze sposobów implementacji sztucznej inteligencji w grach są na przykład automaty skończone bądź logika rozmyta. Automaty skończone lub tzw. „maszyny stanów" to proste mechanizmy mające symulować zachowania danego obiektu. Element rozgrywki jest dzielony na drobne fragmenty, na podstawie których definiuje się stany jako pewne pojedyncze zachowania postaci [5]. Razem ze stanami definiowane są warunki przejścia pomiędzy tymi stanami, odpowiadające jakiejś sytuacji, która musi zaistnieć w grze, aby symulowany obiekt zmienił swoje zachowanie. Jest to dobra opcja do zasymulowania prostych przeciwników, których działania mogą zostać rozłożone do kilku, kilkunastu pojedynczych stanów wraz z prostą reakcją na daną sytuację w grze. Logika rozmyta z kolei odpowiada już bardziej złożonym rozumowaniem sztucznej inteligencji. Opiera się ona na spostrzeżeniu, iż klasyczna dwuwartościowa logika rzadko kiedy odzwierciedla całość ludzkiego rozumienia zewnętrznego świata. Bardzo często jakiś obiekt może być częściowo w danym stanie, a częściowo w innym. Temperatura na zewnątrz może być odczuwana nie tylko jako „gorąco" bądź „zimno". Można ją określić takimi słowami jak np. „chłodno", „ciepło", „umiarkowanie". Do tego dochodzi subiektywny kontekst – dla jednej osoby kilometr drogi może być bardzo dużą wartością, dla innej niekoniecznie. Wszystko zależy od tego w jakiej sytuacji dane osoby się znalazły. Logika rozmyta rozszerza klasyczną logikę o możliwość

bycia w kilku stanach jednocześnie i definiuje przynależność do danego stanu. Tego rodzaju sztuczna inteligencja jest w stanie przeprowadzić wnioskowanie na podstawie danych parametrów wejściowych, dzięki którym potrafi stwierdzić swoją przynależność do poszczególnych stanów i na tej podstawie odpowiednio skorygować swoje zachowanie w czasie. Na Rysunku 1.2 został przedstawiony przykład logiki rozmytej. Obiekt posiada trzy stany (krytyczny, ranny, zdrowy) i na podstawie punktów swojego życia (0-100) określa przynależność do danego stanu.



Rysunek 1.2: Przykład logiki rozmytej

1.3. Silniki gier

W obecnych czasach, projektowanie zaawansowanych gier bez użycia specjalistycznych silników gry praktycznie nie występuje. Istnieje co prawda mnóstwo bibliotek do poszczególnych języków programowania, służących do obsługi takich rzeczy jak fizyka, grafika 2D lub 3D czy audio, ale w znacznej większości firm tworzących gry służą one właśnie do pisania własnych, spersonalizowanych silników, na których później projektanci i programiści pracują. Silnik gry można rozumieć właśnie jako taki zestaw bibliotek wraz ze zintegrowanym środowiskiem programistycznym, służących do szeroko pojętego *game developmentu*. Z najważniejszych rzeczy, które silnik gry powinien mieć wbudowane i móc obsługiwać są m.in. fizyka gry, obsługa grafiki 2D/3D, detekcję kolizji, możliwość pisania skryptów czy zarządzanie pamięcią.

Na rynku obecnie jest wysyp różnych silników gry. Głównie dlatego, że każda firma tworząca gry zazwyczaj używa własnych, spersonalizowanych narzędzi pod konkretne produkty, które planuje wypuścić. Nie zmienia to faktu, że istnieją również opcje darmowe, tworzone przez osobne firmy niezwiązane bezpośrednio z produkcją gier, dostarczające natomiast różne narzędzia i ułatwienia dla projektantów. Z darmowych opcji, które warto rozważyć na pewno są to silniki gier Unity, Unreal Engine oraz w mniejszym stopniu Godot, które zostaną scharakteryzowane poniżej.

1.3.1. Godot

Godot z tej trójki jest silnikiem gry najświeższym, bo wydanym publicznie dopiero w roku 2014. Jest on również jedynym przedstawionym tu silnikiem dostępnym całkowicie za darmo, na licencji MIT. Silnik graficzny Godota obsługuje OpenGL ES w wersji 3.0 oraz moduł graficzny podobny do GLSL. Posiada on również moduły obsługujące grafikę 2D, system zaawansowanych animacji, silnik fizyki Bullet, obsługę audio czy post-processing. Skrypty w Godocie pisane są głównie w natywnym języku GDScript, stworzonym przez twórców Godota na bazie Pythona. Silnik obsługuje również języki C++ oraz C#. Godot działa na Windowsie, Linuxie i macOS oraz wspiera tworzenie gier na platformy pecetowe, konsolowe, mobilne oraz internetowe.

1.3.2. Unreal Engine

Unreal Engine to zdecydowanie najbardziej zaawansowany kandydat. Znany jest z posiadania najbardziej rozwiniętego silnika graficznego 3D, zawierającego mnóstwo bibliotek oraz wspierającego zaawansowane technologie, pozwalającego na niemal fotorealistyczną grafikę w swoich produkcjach. Z racji bycia wspieranym od ponad 16 lat przy dużym nakładzie finansowym przez twórców, jest on chętnie wybieranym silnikiem przez firmy, które nie decydują się na tworzenie własnych, spersonalizowanych narzędzi. Oprócz tego, silnik wspiera standardowe moduły grafiki 2D, obsługi audio, wsparcie dla sztucznej inteligencji oraz silnik fizyczny PhysX stworzony przez NVIDIA. Skryptowanie w Unrealu odbywa się za pomocą języka C++ oraz systemu wizualnego skryptowania zwanego *blueprintami*, który pozwala na tworzenie kodu za pomocą bloczków. Korzystając z tego silnika gry mamy darmowy dostęp do biblioteki Quixel Megascans, która zawiera mnóstwo wysokojakościowych tekstur i assetów umożliwiających tworzenie obiektów w grze. Tworzenie oprogramowania w Unrealu jest darmowe, do momentu przekroczenia pułapu zysków rzędu 1 miliona dolarów. Od tego czasu Unreal pobiera symboliczną opłatę 5% od przyszłych przychodów. Unreal działa na wszystkich najpopularniejszych systemach operacyjnych i za jego pomocą można tworzyć gry na większość obecnie używanych platform.

1.3.3. Unity

Unity to na pewno najbardziej interesujący kandydat ze wszystkich. Powstał w 2005 roku, a twórcom silnika przyświecał jeden cel – umożliwienie większej liczbie programistów narzędzi do tworzenia gier. Unity pozwala na tworzenie gier 2D i 3D oraz udostępnia zaawansowane technologie, takie jak mapowanie tekstur, odbicia, ambient occlusion, potok graficzny HDPR oraz mnóstwo możliwości post-processingu. Nie jest on jednak aż tak zaawansowany graficznie jak jego poprzednik, Unreal Engine. Pomimo tego, jest silnikiem zdecydowanie najłatwiejszym do nauki i tworzenia gier. Z powodu wyżej wymienionej łatwości użycia Unity jest zdecydowanym faworytem dla początkujących twórców gier i niezależnych firm

tworzących raczej mniejsze, budżetowe gry, w tym gry mobilne czy gry rozszerzonej i wirtualnej rzeczywistości AR/VR. Unity wspiera również takie moduły jak obsługa audio, animacji, wbudowany, własny silnik fizyczny oparty na PhysX oraz wsparcie dla sztucznej inteligencji. Unity posiada model subskrypcyjny, gdzie najniższy poziom przeznaczony do nauki i małych komercyjnych projektów jest całkowicie darmowy. Gdy nasze projekty przekroczą roczny pułap 100 tysięcy dolarów przychodów, Unity będzie wymagało od nas przejścia na wyższy, płatny poziom subskrypcji. Pisanie skryptów w Unity odbywa się za pomocą języka C#. Silnik gry jest dostępny na systemy operacyjne Windows, Linux oraz macOS i wspiera wydawanie gier na ponad 19 platform, w tym platformy pecetowe, internetowe, konsolowe, mobilne czy wirtualnej rzeczywistości.

Biorąc pod uwagę te trzy silniki gier zdecydowałem się na użycie silnika Unity do zaprojektowania mojej aplikacji. Czynnikami decydującymi były na pewno moduły wsparcia dla sztucznej inteligencji oraz niski próg wejścia do nauki. Unity ze wszystkich wymienionych wyżej silników gry posiada również najbardziej rozwiniętą dokumentację oraz społeczność skupioną wokół niego, co pomoże szybciej rozwiązać problemy, na które natknę się w trakcie tworzenia i implementacji. Dodatkowym plusem jest również skryptowanie w języku C#, który jest językiem już dobrze mi znanym. W następnym rozdziale zostanie omówiony całościowo projekt systemu walki, jego główne założenia, możliwości dostępne dla gracza oraz zostanie przedstawiona kompleksowo zaprojektowana sztuczna inteligencja przeciwnika.

Rozdział 2

Założenia projektu

2.1. Główne założenia projektu

Tematem pracy licencjackiej jest wykorzystanie metod logiki rozmytej w zachowaniu postaci w grze RPG. Warto byłoby przemyśleć w takim razie całą otoczkę gry, zaplanować w jaki sposób sztuczna inteligencja będzie wykorzystana i w jaki sposób poszczególne elementy zostaną zaimplementowane. Całość projektu i jego głównych założeń jest zawarta w tym podrozdziale.

Projektem pracy jest implementacja prostego systemu walki w klimacie gry fantasy RPG pomiędzy graczem a przeciwnikiem. Zachowanie przeciwnika takie jak poruszanie się bądź wybór ataku jest symulowane sztuczną inteligencją, w której skład wchodzi prosta maszyna stanów oraz logika rozmyta. Walka odbywa się w systemie turowym z pierwszeństwem tury dla gracza. Celem gry jest pokonanie przeciwnika, a sama gra kończy się w momencie pokonania jednej z postaci – przeciwnika przez gracza bądź gracza przez przeciwnika.

Gra jest utrzymana w konwencji 2D z widokiem kamery z góry, tzw. *top-down*. Styl graficzny który został wybrany to styl *pixel-art*, ponieważ jest on dość prosty i łatwo dostępny pod kątem wszelkich grafik na darmowej licencji. W szczególnym przypadku takie grafiki można stworzyć również samodzielnie. Plansza gry jest podzielona kwadratową siatką, względem której odbywa się rozgrywka.

Gra zaczyna się w opuszczonym lesie, gdzie gracz trafia na złego czarnoksiężnika. Plansza gry zawiera takie elementy jak trawa, drzewa, krzewy, kamienie. Każda z postaci może w danej turze wykonać tylko jedną akcję. Na akcję składa się ruch w czterech kierunkach (góra, dół, lewo, prawo) lub atak jedną z czterech dostępnych umiejętności. Ruch jest możliwy w każdym kierunku zawsze, o ile gracz bądź przeciwnik nie koliduje z solidnym blokiem, takim jak kamień bądź drzewo. Możliwość ataku każdą z czterech umiejętności jest rozstrzygana zasięgiem oraz zasobem many. Mana jest terminem używanym w grach fantasy, grach fabularnych i oznacza ona pewnego rodzaju energię, używaną przez magów do czynienia czarów.

Jeśli chodzi o dostępne zasoby, gracz oraz przeciwnik są wyposażeni w zasób życia oraz zasób many. Pierwszy zasób definiuje przeżywalność. Gra kończy się, gdy życie którejś z postaci spadnie do zera. Zasób many determinuje możliwość wykonania ataku, ponieważ każdy z ataków „kosztuje" pewną liczbę many. Gracz jak i przeciwnik posiadają również wcześniej wymienione cztery ataki.

Każda postać zawiera ten sam repertuar zasobów oraz ataków, również pod względem liczbowym. Zasób życia jest ustawiony na wartość 100 punktów, zasób many na wartość punktów 200. Każdy zasób podlega regeneracji o odpowiednio 5 i 15 punktów z początkiem każdej tury. W Tabeli 2.1 zostaną przedstawione cztery możliwe ataki do wykonania.

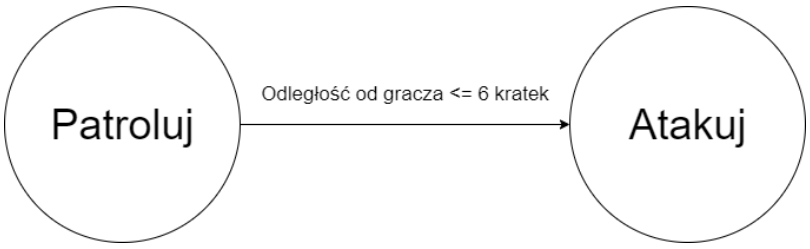
Nazwa	**Zasięg**	**Koszt many**	**Obrażenia**
Kula ognia	5 kratek	60 many	20-30 obrażeń
Pocisk lodu	3 kratki	40 many	15-20 obrażeń
Błyskawica	6 kratek	50 many	10-35 obrażeń
Uderzenie kijem	1 kratka	0 many	5-10 obrażeń

Tabela 2.1: Możliwe ataki do wykonania

2.2. Sztuczna inteligencja przeciwnika

Sztuczna inteligencja przeciwnika stanowi kluczowy element pracy licencjackiej. Jest ona zaimplementowana na wzór prostej maszyny stanów, aby zdeterminować możliwe stany, jakie przeciwnik może przybrać oraz logiki rozmytej, aby określić wybór ataku na podstawie danych w grze.

2.2.1. Maszyna stanów



Rysunek 2.1: Maszyna stanów przeciwnika

Na Rysunku 2.1 został przedstawiony model maszyny stanów przeciwnika. Zaczyna on grę od stanu **Patroluj**. W tym stanie porusza się losowo w którymś z czterech kierunków (góra, dół, lewo, prawo). Warunkiem przejścia do stanu **Atakuj** jest odległość 6 lub mniej kratek od gracza. Warunek jest sprawdzany na początku każdej tury przeciwnika. Stan **Atakuj** jest jednocześnie stanem końcowym

– nie istnieje przejście do żadnego innego stanu, zatem przeciwnik będzie atakował do końca gry. W Tabeli 2.2 znajduje się lista możliwych stanów oraz zachowania przeciwnika, a w Tabeli 2.3 znajdują się wszystkie możliwe przejścia pomiędzy stanami.

Stan	**Zachowanie**
Patroluj	Przeciwnik porusza się losowo w jednym z czterech kierunków
Atakuj	Na podstawie logiki rozmytej przeciwnik dobiera atak

Tabela 2.2: Możliwe stany przeciwnika

Stan pierwotny	**Stan docelowy**	**Warunek**
Patroluj	Atakuj	Odległość mniejsza bądź równa 6 kratek od gracza

Tabela 2.3: Możliwe przejścia pomiędzy stanami przeciwnika

Stan **Atakuj** cechuje się zaimplementowaną logiką rozmytą w celu zdeterminowania jednego z czterech ataków.

2.2.2. Logika rozmyta

Istotnym elementem podczas projektowania logiki rozmytej do wyboru ataków przez przeciwnika jest wybór sytuacji, elementów, które przeciwnik będzie brał pod uwagę i dostosowywał swoje zachowanie. Należy określić kilka decydujących czynników, na które składałby się ostateczny wybór. Przeanalizowania wymagają więc dostępne ataki zaprojektowane wcześniej. Jest ich do wyboru cztery, z czego każdy z nich cechuje się wybranym zasięgiem, kosztem many oraz obrażeniami. Elementem warunkującym, czy dany atak jest wykonalny jest zasięg oraz koszt many. Jeżeli przeciwnik nie będzie posiadał odpowiedniego zasięgu, bądź liczba many, którą posiada będzie zbyt mała, przeciwnik nie będzie mógł go wykonać. Pewnym specyficznym atakiem jest atak **Uderzenie kijem**, ponieważ nie wymaga on w ogóle kosztu many, natomiast posiada bardzo restrykcyjny zasięg, gdyż przeciwnik musi stać obok gracza, w obrębie co najwyżej jednej kratki. Istotnym czynnikiem jest to, że rozstrzał obrażeń pomiędzy atakami również ma znaczenie. Najmniejszy rozstrzał w obrażeniach posiada atak **Pocisk lodu**, gdyż jest to tylko **15-20**. Używając tego ataku, przeciwnik może być w miarę pewny jakiego rzędu obrażenia zada, niestety kosztem dość krótkiego zasięgu oraz niskiej liczby obrażeń. Z drugiej strony zaś, atak **Błyskawica** ma rozstrzał obrażeń aż **10-35**, co sprawia, że element losowy ma tutaj o wiele większe znaczenie. Zatem w celu zbalansowania rozgrywki przy większej losowości, atak ten posiada największy zasięg ze wszystkich możliwych.

Ważnym elementem rozgrywki jest to, aby przeciwnik zachowywał się w miarę racjonalnie, czyli pilnował swoich zasobów, aby się nie wyczerpały oraz dobierał atak na podstawie swoich możliwości i sytuacji. Duże znaczenie ma koszt many

oraz rozstrzał obrażeń, zatem dobór ataku będzie uzależniony od tych dwóch zmiennych. Sztuczna inteligencja będzie sprawdzać ile przeciwnik posiada punktów many oraz ile gracz posiada punktów życia i na tej podstawie oszacowuje każdy atak w skali liczbowej, czyli tzw. przynależność do zbioru w definicji logiki rozmytej. Jeżeli tak wybrany atak okaże się niemożliwy do wykonania ze względu na zbyt dużą odległość, zamiast zaatakować, przeciwnik w danej turze wykona ruch w stronę gracza. Z tego powodu atak **Uderzenie kijem** jest wyłączony z szacowania, ponieważ zasięg (1 kratka) jest zbyt restrykcyjny. W zamian za to ten atak będzie wybierany zawsze w momencie, w którym przeciwnik będzie posiadał mniej niż 40 punktów many, gdyż w takiej sytuacji reszta ataków nie byłaby możliwa do wykonania.

Każdy atak podlega ewaluacji na dwóch płaszczyznach – tego, ile przeciwnik posiada punktów many oraz tego, ile gracz posiada punktów życia. Z tego powodu każda ewaluacja będzie odbywała się w skali 0-50, aby po połączeniu dostać finalnie wartość z zakresu 0-100 przy każdym ataku. Warto sobie również wypisać pewien zestaw reguł, którymi sztuczna inteligencja powinna się kierować, a następnie odpowiednio dobrać do tego wartości liczbowe.

1. Jeżeli przeciwnik ma mniej niż 40 punktów many, jego jedynym wyborem będzie atak **Uderzenie kijem**, ponieważ nie jest w stanie wykonać żadnego innego ataku.

2. Jeżeli przeciwnik ma niewielką liczbę punktów many, jego głównym wyborem będzie atak **Pocisk lodu**, ponieważ istotnym jest, aby przeciwnik oszczędzał punkty many.

3. Jeżeli przeciwnik ma średnią liczbę punktów many, jego głównym wyborem będzie atak **Błyskawica**, ponieważ jest w stanie pozwolić sobie na kosztowniejszy atak zadający większe obrażenia.

4. Jeżeli przeciwnik ma dużą liczbę punktów many, jego głównym wyborem będzie atak **Kula ognia**, ponieważ zadaje on największe obrażenia.

5. Jeżeli gracz posiada mniej niż 10 punktów życia, każdy atak będzie wartościowany tak samo, ponieważ każdy atak będzie zabójczy.

6. Jeżeli gracz posiada mało punktów życia, wysoko wartościowanymi atakami będą ataki **Kula ognia** oraz **Pocisk lodu**, ponieważ są najbardziej solidne w kontekście zadawanych obrażeń.

7. Jeżeli gracz posiada dużo punktów życia, wysoko wartościowanymi atakami będą ataki **Błyskawica** oraz **Kula ognia**, ponieważ mają wysokie obrażenia.

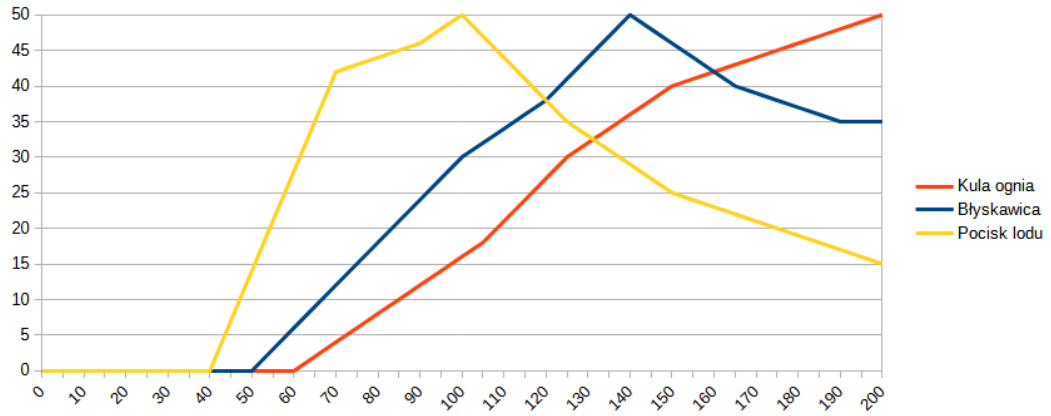
Na tej podstawie zostały dobrane odpowiednie wartości liczbowe wszystkich ataków. W Tabeli 2.4 przedstawione zostały dokładne wartości liczbowe dla punktów many przeciwnika (wartości 0-200), a w Tabeli 2.5 wartości liczbowe dla punktów zdrowia gracza (wartości 0-100).

Punkty many przeciwnika	Błyskawica	Kula ognia	Pocisk lodu
0	0	0	0
5	0	0	0
10	0	0	0
15	0	0	0
20	0	0	0
25	0	0	0
30	0	0	0
35	0	0	0
40	0	0	0
45	0	0	7
50	0	0	14
55	3	0	21
60	6	0	28
65	9	2	35
70	12	4	42
75	15	6	43
80	18	8	44
85	21	10	45
90	24	12	46
95	27	14	48
100	30	16	50
105	32	18	47
110	34	21	44
115	36	24	41
120	38	27	38
125	41	30	35
130	44	32	33
135	47	34	31
140	50	36	29
145	48	38	27
150	46	40	25
155	44	41	24
160	42	42	23
165	40	43	22
170	39	44	21
175	38	45	20
180	37	46	19
185	36	47	18
190	35	48	17
195	35	49	16
200	35	50	15

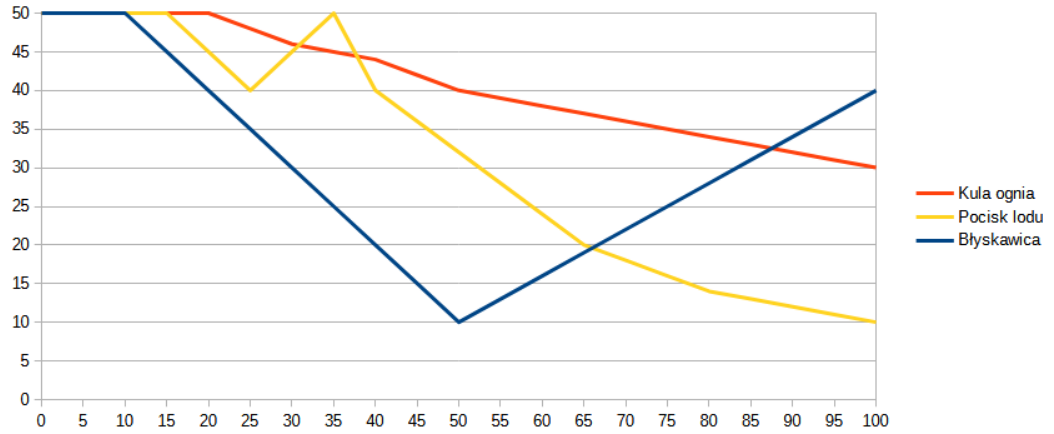
Tabela 2.4: Wartości liczbowe poszczególnych ataków dla punktów many przeciwnika

Punkty życia gracza	Błyskawica	Kula ognia	Pocisk lodu
0	50	50	50
5	50	50	50
10	50	50	50
15	45	50	50
20	40	50	45
25	35	48	40
30	30	46	45
35	25	45	50
40	20	44	40
45	15	42	36
50	10	40	32
55	13	39	28
60	16	38	24
65	19	37	20
70	22	36	18
75	25	35	16
80	28	34	14
85	31	33	13
90	34	32	12
95	37	31	11
100	40	30	10

Tabela 2.5: Wartości liczbowe poszczególnych ataków dla punktów życia gracza



Rysunek 2.2: Wykres ewaluacji trzech ataków przeciwnika na podstawie jego punktów many



Rysunek 2.3: Wykres ewaluacji trzech ataków przeciwnika na podstawie punktów życia gracza

Na Rysunkach 2.2 i 2.3 zostały przedstawione dwa wykresy ewaluacji każdego z trzech ataków przeciwnika, na podstawie danych wejściowych. W zależności od sytuacji, każdy atak ma przypisaną wartość liczbową z zakresu 0-100 i określa jego przynależność do zbioru. Aby gra była ciekawsza, a wybór ataku nie był determinowany tylko do tych dwóch czynników, ostateczny wybór ataku będzie odbywał się na drodze losowania liczby z trzech zbiorów. Przynależność do zbioru będzie skorelowana tylko z prawdopodobieństwem wyboru. Mianowicie, po obliczeniu trzech wartości dla każdego ataku są tworzone trzy zbiory liczb o liczebności wartości poszczególnych ataków, a następnie losowana jest liczba z przedziału od 0 do sumy wszystkich wartości. Wylosowana liczba musi należeć do któregoś zbioru, a więc wybranym atakiem będzie atak z wartością równą liczebności wylosowanego zbioru. Za przykład niech posłużą przedstawione poniżej dwie sytuacje:

1 sytuacja:
Przeciwnik posiada 65 punktów many, a gracz posiada 20 punktów życia

1. Wartość ataku **Błyskawica** dla punktów many przeciwnika 65 wynosi **9**, a dla punktów życia gracza 20 – **40**. Zatem łączna wartość dla ataku **Błyskawica** wynosi **49**.
2. Wartość ataku **Kula ognia** dla punktów many przeciwnika 65 wynosi **2**, a dla punktów życia gracza 20 – **50**. Zatem łączna wartość dla ataku **Kula ognia** wynosi **52**.
3. Wartość ataku **Pocisk lodu** dla punktów many przeciwnika 65 wynosi **35**, a dla punktów życia gracza 20 – **45**. Zatem łączna wartość dla ataku **Pocisk lodu** wynosi **80**.
4. Zsumowane ataki:
   (a) Błyskawica: 49
   (b) Kula ognia: 52
   (c) Pocisk lodu: 80
5. Tworzone są trzy zbiory, każdy o liczebności danego ataku:
   (a) Zbiór A: Wartości od 0 do 48 włącznie (Liczebność: 49)
   (b) Zbiór B: Wartości od 49 do 100 włącznie (Liczebność: 52)
   (c) Zbiór C: Wartości od 101 do 180 włącznie (Liczebność: 80)
6. Losowana jest liczba z przedziału 0-180 włącznie
7. Wylosowaną liczbą jest 132 – należy ona do zbioru C
8. Zatem wybranym atakiem jest atak **Pocisk lodu**

2 sytuacja:
Przeciwnik posiada 125 punktów many, a gracz posiada 90 punktów życia

1. Wartość ataku **Błyskawica** dla punktów many przeciwnika 125 wynosi **41**, a dla punktów życia gracza 90 – **34**. Zatem łączna wartość dla ataku **Błyskawica** wynosi **75**.

2. Wartość ataku **Kula ognia** dla punktów many przeciwnika 125 wynosi **30**, a dla punktów życia gracza 90 – **32**. Zatem łączna wartość dla ataku **Kula ognia** wynosi **62**.

3. Wartość ataku **Pocisk lodu** dla punktów many przeciwnika 125 wynosi **35**, a dla punktów życia gracza 90 – **12**. Zatem łączna wartość dla ataku **Pocisk lodu** wynosi **47**.

4. Zsumowane ataki:

   (a) Błyskawica: 75

   (b) Kula ognia: 62

   (c) Pocisk lodu: 47

5. Tworzone są trzy zbiory, każdy o liczebności danego ataku:

   (a) Zbiór A: Wartości od 0 do 74 włącznie (Liczebność: 75)

   (b) Zbiór B: Wartości od 75 do 136 włącznie (Liczebność: 62)

   (c) Zbiór C: Wartości od 137 do 183 włącznie (Liczebność: 47)

6. Losowana jest liczba z przedziału 0-183 włącznie

7. Wylosowaną liczbą jest 11 – należy ona do zbioru A

8. Zatem wybranym atakiem jest atak **Błyskawica**

Powyższe sytuacje pokazały w jaki sposób sztuczna inteligencja będzie dokonywać wyboru ataku na podstawie sytuacji w grze. W pierwszej sytuacji możemy zauważyć, że liczba punktów many przeciwnika jest dość niska, zatem bardziej wartościowanym atakiem jest atak **Pocisk lodu**. W drugim przypadku przeciwnik posiada o wiele więcej punktów many oraz gracz jest praktycznie zdrowy – w tym wypadku bardziej wartościowanymi atakami są ataki **Kula ognia** oraz **Błyskawica**.

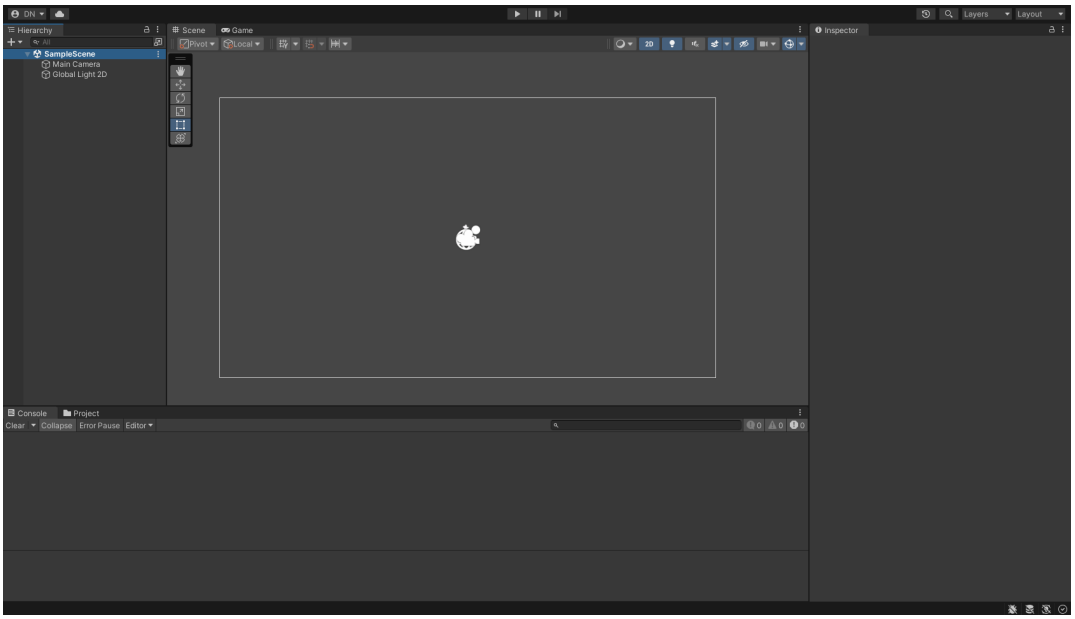
Rozdział 3

Implementacja

W tym rozdziale zostanie przedstawiona implementacja całej gry, zostaną pokazane kawałki kodu odpowiedzialne za poszczególne elementy, razem z wycinkami scen z gry oraz wyjaśnieniem, w jaki sposób zostały zaimplementowane.

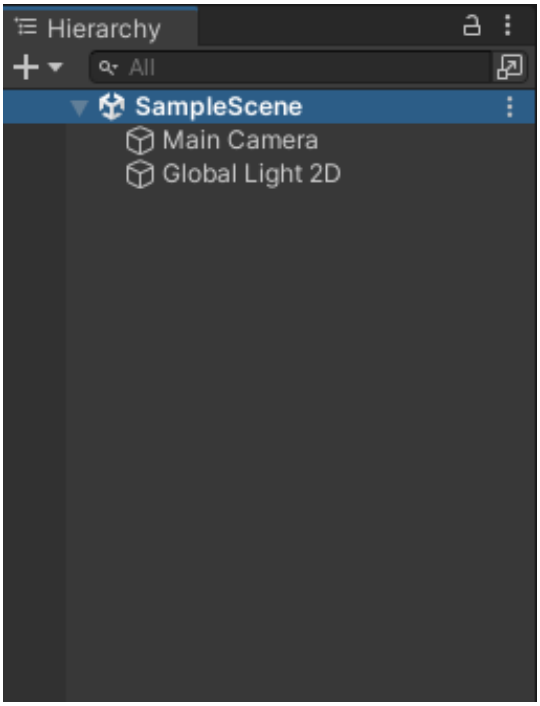
3.1. Przygotowanie i omówienie środowiska

Aplikacja została stworzona na wersji Unity 2022.3.7f1, która jest wersją LTS *(ang. Long Term Support)*. Posiada ona niezbędne moduły potrzebne do realizacji projektu, jest stabilna i posiada długoterminowe wsparcie. Dzięki temu nie było większych problemów z samym narzędziem w trakcie implementacji. Wersja ta została wyposażona również w możliwość budowania aplikacji na systemy Windows oraz Linux. Całość jest umieszczona na moim githubie, aby w każdej chwili można było sobie ją przetestować. Językiem skryptowym oczywiście będzie język C#.



Rysunek 3.1: Główne okno silnika gry Unity

Główny interfejs silnika Unity zawiera kilka różnych okienek, które zostaną omówione poniżej. Na Rysunku 3.1 można zobaczyć jak prezentuje się całe okno Unity.

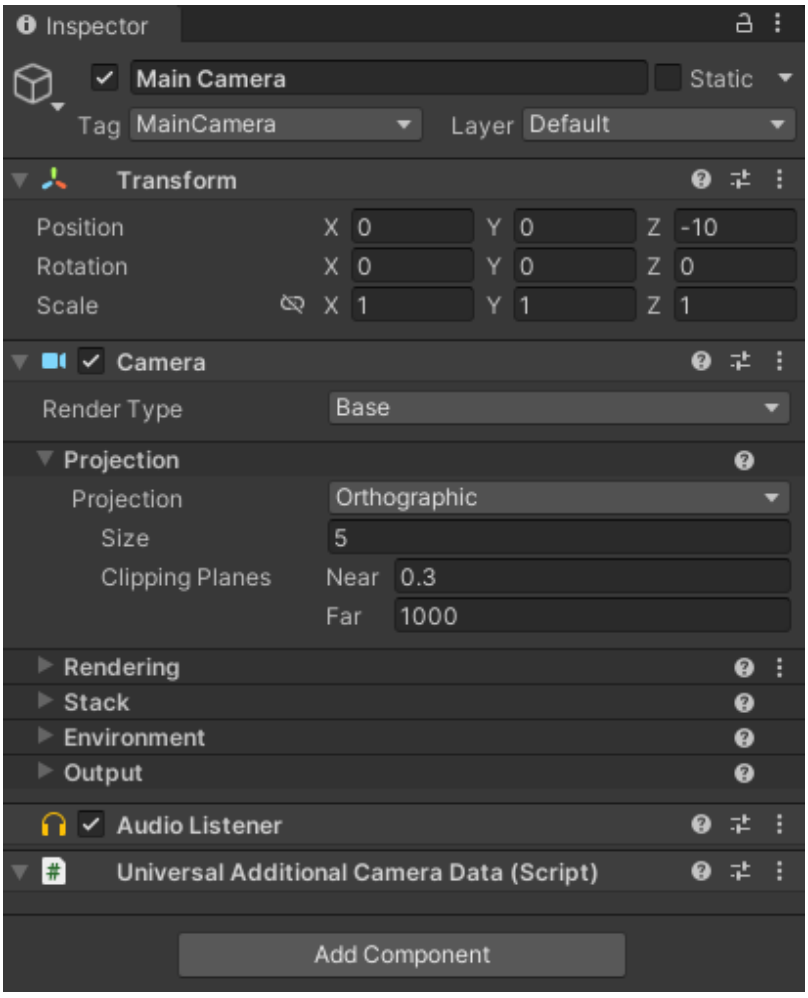


Rysunek 3.2: Okienko hierarchii sceny

Po lewej stronie znajduje się okno nazwane hierarchią *(ang. hierarchy)*. Jest ono przedstawione na Rysunku 3.2. Zawiera ono nazwę sceny oraz wszystkie elementy, które są obecne na tej scenie w postaci wylistowanej. Domyślnie Unity dzieli całość gry na tzw. sceny. Scena to po prostu element, który widać od razu po włączeniu gry. Unity pozwala tworzyć wiele scen, przełączać się między nimi oraz umieszczać na nich dowolne obiekty. Przed uruchomieniem gry, każdy element na scenie musi zostać przedtem załadowany do pamięci gry, aby całość mogła się wyświetlić na ekranie, zatem dobrą praktyką jest nieprzeładowywanie scen dużą liczbą elementów. Dla małych gier nie jest to większym problemem, dla większych warto dbać o to, aby dzielić grę na poszczególne sceny, na przykład wykorzystywać oddzielne sceny dla różnych poziomów gry czy lokacji.

Na każdej nowo utworzonej scenie znajdują się domyślnie dwa obiekty: kamera oraz światło. Hierarchia pomaga w szybkim zorientowaniu się w tym, jakie elementy znajdują się na naszej scenie. Postać listy pomaga również w pokazaniu relacji rodzic-dziecko pomiędzy poszczególnymi obiektami.

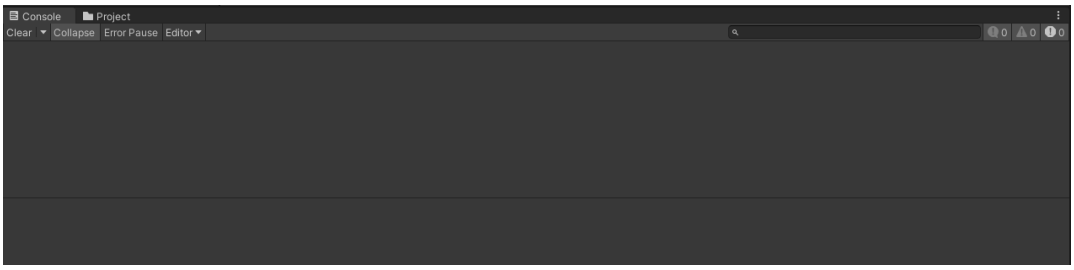
Na samym środku widnieją dwa okienka – sceny oraz gry, pomiędzy którymi można się przełączać. Okienko sceny pokazuje wizualnie wszystkie elementy, które są obecne na scenie, a okienko gry pokazuje jak te elementy będą wyglądały po uruchomieniu aplikacji. Nie wszystko obecne na scenie będzie zawsze widoczne w grze. Odpowiada za to obiekt kamery. Dlatego też na scenie widać duży prostokąt obejmujący jego część. Na samej scenie można przestawiać, dodawać, usuwać, zmieniać rozmiary różnym elementom w grze, jest to swego rodzaju „piaskownica". Nad okienkiem sceny znajdują się także dodatkowe przyciski startu, pauzy i kroku. Służą one do testowania aplikacji. W momencie naciśnięcia przycisku Start, okno sceny przełącza się na okno gry i można zobaczyć jak gra będzie wyglądała z poziomu uruchomienia jej.



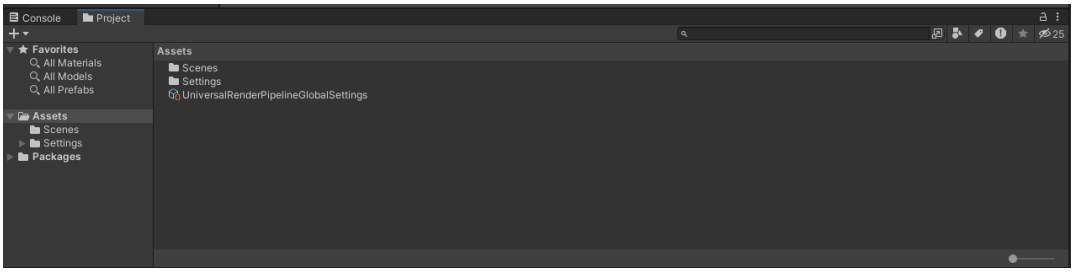
Rysunek 3.3: Okienko inspektora

Po prawej stronie znajduje się okienko inspektora *(ang. inspector)*, widoczne na Rysunku 3.3. Odpowiada ono za wyświetlenie wszystkich właściwości i komponentów danego elementu obecnego na scenie gry. Tutaj dla przykładu zaznaczony został obiekt kamery, aby okienko inspektora pokazało jej właściwości. Można tutaj zmieniać danemu obiektowi nazwę, tag lub aktywność, dodawać lub usuwać komponenty oraz zmieniać ich wartości. Domyślnie każdy nowo utworzony element

na scenie ma komponent nazwany *Transform*. Odpowiada on za położenie, rotację oraz skalę obiektu gry w osiach X, Y oraz Z. Wszystkie skrypty, które kiedykolwiek zostaną napisane również są podpinane i wyświetlane tutaj. Jeśli chodzi o sam element kamery, to oprócz komponentu *Transform* posiada on również komponenty *Camera*, *Audio Listener* oraz skrypt domyślnie podpięty przez Unity. Komponent *Camera* odpowiada za właściwości kamery – można tutaj zmienić odległość kamery, typ renderowania czy projekcję na perspektywiczną bądź ortogonalną. Komponent *Audio Listener* odpowiada za odtwarzanie dźwięku w grze, a dołączony automatycznie skrypt odpowiada za pewne właściwości i zachowania kamery w grze.



Rysunek 3.4: Okienko konsoli



Rysunek 3.5: Okienko projektu

Ostatnimi okienkami, jakie zostaną omówione to okienka położone na samym dole, pomiędzy którymi można się przełączać. Są to okienka konsoli oraz projektu. Okienko konsoli (Rysunek 3.4) jest przydatne podczas skryptowania, ponieważ wyświetla wszelkie ostrzeżenia lub błędy, które zostały popełnione w trakcie pisania. Przydaje się ono również podczas testowania gry, gdyż wszelkie błędy w trakcie działania gry również wyświetlają się w konsoli. Okienko projektu (Rysunek 3.5) służy do wyświetlenia wszystkich elementów, które obecnie zawiera nasz projekt, a które niekoniecznie są ustawione w danym momencie na scenie. Można tu zarządzać katalogami, tworzyć i dołączać wszelkie elementy takie jak grafiki, skrypty, dźwięki czy sceny. Każdy element zanim pojawi się na scenie, musi wcześniej zostać dodany do projektu.

Warto wspomnieć o tym, że interfejs Unity jest wysoce konfigurowalny. Każde okienko wymienione wyżej można przenosić, skalować, zmieniać jego położenie, wszystko wedle własnego uznania. Dodatkowo można włączać i dodawać różne inne okienka, których Unity domyślnie nam nie wyświetla. Jednym z takich

przykładów jest okienko do tworzenia animacji, które zostało wykorzystane w trakcie implementacji projektu. Do dołączania okienek służy menu kontekstowe *Window* na samej górze interfejsu Unity.

Na samym końcu zostanie pokrótce przedstawione jak wygląda skryptowanie w Unity. Każdy nowo utworzony skrypt wygląda jak na Listingu 3.1.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class TestScript : MonoBehaviour
6 {
7     // Start is called before the first frame update
8     void Start()
9     {
10
11    }
12
13    // Update is called once per frame
14    void Update()
15    {
16
17    }
18 }
```

Listing 3.1: Przykładowy wygląd skryptu TestScript.cs

Na samym początku mamy trzy dyrektywy *using*. Sa one odpowiedzialne za zaciągnięcie odpowiednich bibliotek Unity. Następnie mamy deklarację klasy, o nazwie takiej samej jak nazywa się skrypt (w tym przypadku skrypt został nazwany „TestScript") oraz dziedziczenie po klasie *MonoBehaviour*. Klasa MonoBehaviour zawiera definicję funkcji sterujących „cyklem życia" w grze. Domyślnie po utworzeniu skryptu istnieją dwie takie funkcje – *Start* oraz *Update*. Funkcja Start wywołuje się tylko raz, na samym początku gry. Funkcja Update natomiast wywołuje się co każdą klatkę gry. Są to bardzo przydatne funkcje, gdy chcemy sterować różnymi elementami lub obiektami w grze w czasie.

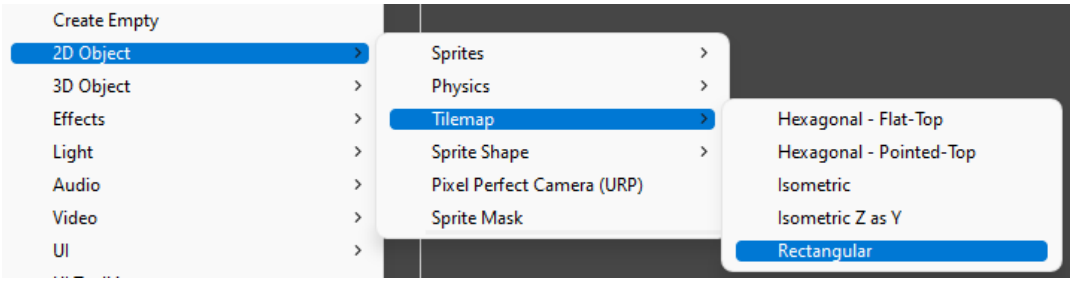
W zasadzie w prawie każdym przypadku na jeden skrypt przypada tylko jedna klasa, nazwana tak samo jak skrypt. Jest możliwa deklaracja kilku klas w obrębie jednego skryptu, jednak odradza się takie rozwiązanie ze względu na czytelność kodu. W dalszej części pracy słowa skrypt i klasa będą używane zamiennie, pamiętając o tym, aby w jednym skrypcie była umieszczana tylko jedna deklaracja klasy. Będzie to przydatne w trakcie tworzenia referencji do innych klas/skryptów w obrębie obecnie tworzonego skryptu.

3.2. Plansza gry, siatka i postacie

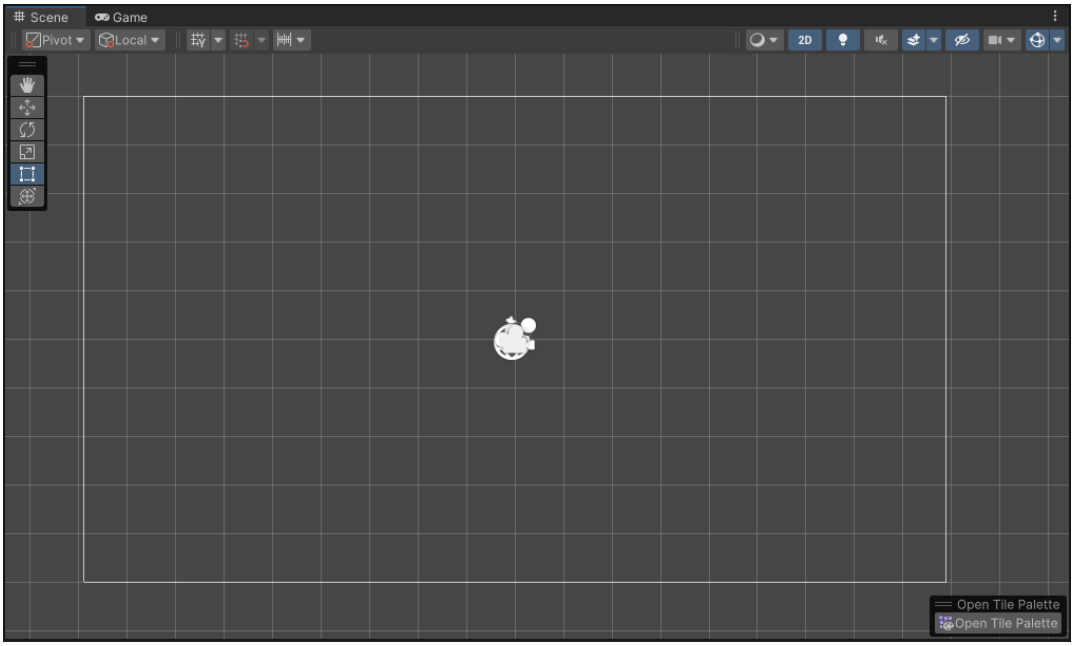
Na samym starcie przedstawiony zostanie wygląd planszy gry, w jaki sposób plansza została stworzona i jak działa siatka gry. We wcześniejszym rozdziale wspomniane było, że gra ma charakter 2D pixel-art, a plansza gry jest podzielona siatką, względem której są ustawione różne elementy otoczenia i względem której odbywa się rozgrywka.

W Unity można utworzyć obiekt nazwany *Grid*, który tworzy siatkę na planszy gry i pozwala odpowiednio wstawiać pojedyncze płytki gry *(ang. tiles)*. Typów siatki w Unity jest kilka, na przykład siatka kwadratowa, siatka heksagonalna czy izometryczna. Na potrzeby projektu użyta została siatka kwadratowa, gdzie pojedynczymi „płytkami" są kwadraty.

Aby móc zapełnić planszę gry *sprite'ami*, czyli pojedynczymi, pikselowymi obrazkami 2D, musi zostać stworzony kolejny obiekt nazwany *tilemapą*. Jest to element, który przechowuje wszystkie narysowane sprite. Tilemap można stworzyć wiele, a każda może odpowiadać za oddzielny element „mapy" i do każdej można dołączyć oddzielne komponenty w inspektorze. Na Rysunku 3.6 można zobaczyć menu kontekstowe tworzenia obiektu Grid razem z tilemapą, a na Rysunku 3.7 jak nowo utworzony obiekt Grid wygląda na scenie projektu. Do tworzenia mapy wykorzystana została paczka assetów Pixel Art Top Down – Basic twórcy Cainos na darmowej licencji [6].



Rysunek 3.6: Menu kontekstowe tworzenia tilemapy – po wybraniu odpowiedniej opcji obiekt Grid tworzy się automatycznie



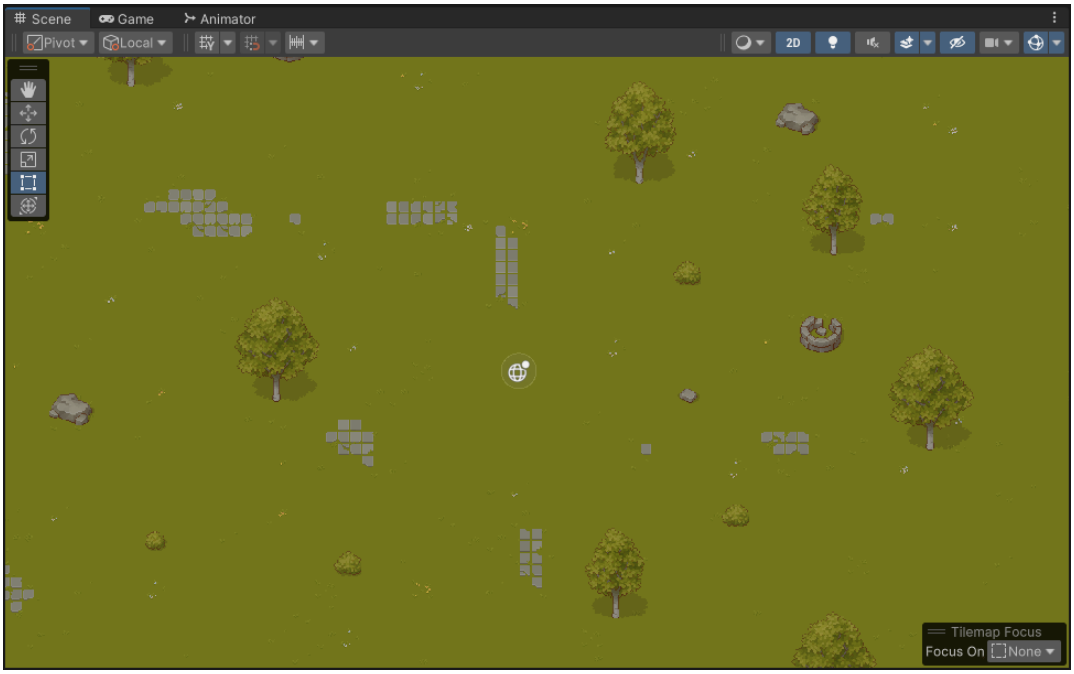
Rysunek 3.7: Obiekt Grid widoczny na scenie – plansza gry została podzielona na kwadraty, które następnie można wypełnić odpowiednimi sprite'ami

W paczce assetów wykorzystane zostały takie elementy jak trawa, drzewa, kamienie czy krzewy do stworzenia prostej mapy 2D. Warto było więc zadbać o odpowiednie rozgraniczenie tych elementów. Gracz powinien poruszać się po trawie, ale również nie powinien mógł wchodzić w drzewa czy kamienie. W wyniku czego mapa gry składa się z kilku tilemap, gdzie niektóre z nich posiadają odpowiednie komponenty.

Rysunek 3.8: Tilemapy dostępne w grze

Na Rysunku 3.8 widać zaimplementowane cztery tilemapy nazwane po kolei *Trees*, *Rocks*, *Shadows* oraz *Background* (odpowiednio: drzewa, kamienie, cienie oraz tło). Każda z tej tilemapy zawiera odpowiednio narysowane sprite na planszy gry. Tilemapa *Trees* odpowiada za drzewa i krzewy, tilemapa *Rocks* odpowiada za kamienie, tilemapa *Shadows* odpowiada za cienie, a tilemapa *Background* odpowiada za tło – w tym wypadku elementy otoczenia, po których gracz może chodzić, takie jak trawa, kwiatki, ścieżka. Dodatkowo, do tilemapy odpowiadającej za drzewa, krzewy i kamienie został dodany w inspektorze komponent *Tilemap Collider 2D*. Jest to komponent odpowiadający za dodanie kolizji do wszystkich

elementów narysowanych na tilemapie. Tak jak wcześniej było wspomniane, gracz nie powinien móc chodzić swobodnie po takich elementach jak drzewa czy kamienie, więc ten komponent jest potrzebny, aby móc wykrywać kolizje pomiędzy graczem a elementami tilemapy. Na Rysunku 3.9 można zobaczyć wycinek mapy stworzonej w grze. Jeśli chodzi o postacie do gry zostały one stworzone za pomocą darmowego generatora postaci LPC [7].



Rysunek 3.9: Wycinek mapy stworzonej w grze

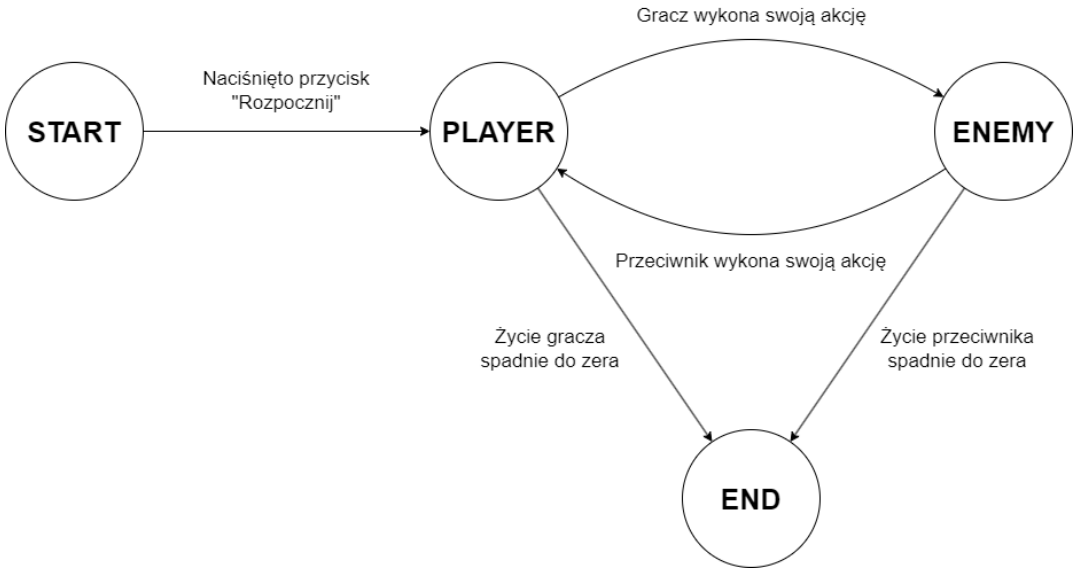
Rysunek 3.10: Postacie gracza (po lewej) oraz przeciwnika (po prawej)

Postacie gracza oraz przeciwnika (widoczne na Rysunku 3.10) będą zaimplementowane jako oddzielne obiekty gry *(ang. game object)*, aby można było dołączać do nich odpowiednie skrypty i komponenty, tak samo jak w przypadku oddzielnych tilemap. Do obiektu gracza będzie dodatkowo dołączony obiekt głównej kamery, gdyż chcę, aby kamera poruszała się razem z graczem.

3.3. System turowy i menedżer gry

Przy implementacji różnorakich systemów lub mechanik gry, które będą oddziaływały na nią w trakcie, warto stworzyć sobie pewien specyficzny obiekt o nazwie menedżera gry *(ang. Game Manager)*. Będzie to niewidzialny obiekt, którego zadaniem będzie przechowywanie skryptu o tej samej nazwie, w którym zawarte są wszelkie elementy potrzebne do prawidłowego działania gry. W poprzednim rozdziale było wspomniane o tym, że gra będzie odbywała się w systemie turowym, z pierwszeństwem ruchu gracza. Takowy system jest idealnym przykładem, aby zawrzeć go w skrypcie menedżera gry.

Gra powinna mieć zaimplementowaną turę gracza oraz przeciwnika. Warto również pomyśleć o stanie początkowym i końcowym gry, które odpowiadałyby za wyświetlanie i zarządzanie odpowiednimi oknami gry. Po włączeniu aplikacji gra powinna powitać nas menu początkowym, w którym można rozpocząć grę, przeczytać instrukcję albo wyjść z aplikacji. Po kliknięciu guzika „Start" w menu początkowym gra automatycznie powinna przenieść gracza do rozgrywki, w której walczy z przeciwnikiem. Gdy życie którejś z postaci spadnie do zera, gra się zakończy. Powinien zatem zostać wyświetlony odpowiedni ekran o tym, kto zwyciężył wraz z możliwością rozegrania partii ponownie lub wyjścia z aplikacji. Można więc określić cztery podstawowe stany gry – stan początkowy, stan rozgrywki, w który wchodzą tura gracza i tura przeciwnika oraz stan końcowy gry. Zostało to przedstawione na rysunku 3.11.



Rysunek 3.11: Diagram stanów gry

Wiedząc już w jaki sposób ten system powinien zostać zaimplementowany, można omówić skrypt menedżera gry.

System turowy w skrypcie menedżera gry jest obsługiwany przez enumerator **TURN**, który przechowuje cztery możliwe stany gry – START, PLAYER, ENEMY oraz END. Aktualny stan gry będzie przechowywany w prywatnej zmiennej typu TURN, do której został napisany prosty akcesor. W funkcji *Start()* zmienna przechowująca aktualną turę jest ustalona na wartość START, ponieważ jest to pierwszy stan gry po jej uruchomieniu.

Funkcja *SetTurnTo()* pełni rolę rozbudowanego mutatora do zmiennej przechowującej aktualną turę. Jest ona wywoływana w wielu miejscach gry – głównie, gdy gracz lub przeciwnik wykonają akcję i zakończą swoją turę. Jako że istnieją cztery możliwe stany gry, została stworzona prosta konstrukcja *switch case*, w ramach której sprawdzany jest przekazany argument w funkcji i wywoływany odpowiedni kawałek kodu. Przypadki „START" oraz „END" są dość trywialne. O wiele więcej dzieje się w przypadkach obsługi tur gracza oraz przeciwnika. W każdej turze na samym początku sprawdzana jest wartość ich punktów życia. Jeżeli spadną one do zera, tura zostaje zmieniona na wartość END i wywołują się funkcje odpowiadające za obsługę śmierci postaci. Jeżeli natomiast wartości ich punktów życia są większe od zera, wartość zmiennej aktualnej tury zostaje zmieniona na odpowiadającą graczowi lub przeciwnikowi.

W samym skrypcie menedżera gry zostało zaprogramowane jeszcze kilka pomocniczych funkcji, które szerzej omówione zostaną w kolejnych podrozdziałach, gdzie ich funkcja zostanie przedstawiona w szerszym kontekście.

3.4. Poruszanie się postacią i zasoby postaci

Poruszanie się postacią stanowi obowiązkowy element w prawie każdej grze, a tym bardziej grze fabularnej. Możliwość wcielenia się w postać, możliwość sterowania nią, zasoby charakteryzujące ją stanowią immanentny element gier komputerowych. W tym podrozdziale zostaną omówione skrypty gracza oraz przeciwnika, które służą do sterowania postacią oraz skrypty definiujące wcześniej określone zasoby, jakimi postać powinna dysponować.

W poprzednim rozdziale było wspomniane, że gracz oraz przeciwnik w swojej turze mogą wykonać tylko jedną akcję, na którą składa się ruch albo atak. Jako że plansza gry składa się z kwadratowej siatki, ruch, który gracz może wykonać jest limitowany tylko do czterech kierunków, tj. góra, dół, lewo bądź prawo. Istotnym elementem jest to, aby gracz w jednej turze mógł poruszyć się tylko w jedną stronę, o jedną kratkę. Obiekt Grid omówiony w pierwszym podrozdziale podzielił planszę gry na kwadraty, automatycznie dbając o to, aby odległości pomiędzy każdymi kwadratami były równe 1. Dzięki temu poruszanie się gracza zawsze będzie odbywało się o 1 jednostkę odległości w silniku. Warto jeszcze wspomnieć, że gra powinna wyświetlać animację ruchu, oraz obiekt gracza nie powinien przeskakiwać od razu pomiędzy jednostkami, a raczej pokazywać płynny ruch z pozycji obecnej do docelowej w czasie. Dlatego wykorzystanym elementem tutaj będą kurutyny.

Kurutyna to specjalny typ funkcji w Unity, który pozwala na wstrzymanie jej działania przez określony czas. Jest to bardzo przydatne, gdy trzeba na przykład zatrzymać działanie gry na kilka sekund lub wywołać animację prosto ze skryptu. W tym przypadku, użycie kurutyn jest wskazane, gdyż zaimplementowany został płynny ruch obiektu postaci gracza w zadanym czasie.

```
1 public void MoveUp() {
2     StartCoroutine(Move(Vector3.up));
3     animator.SetTrigger("Up");
4 }
5
6 public void MoveDown() {
7     StartCoroutine(Move(Vector3.down));
8     animator.SetTrigger("Down");
9 }
10
11 public void MoveLeft() {
12     StartCoroutine(Move(Vector3.left));
13     animator.SetTrigger("Left");
14 }
15
16 public void MoveRight() {
17     StartCoroutine(Move(Vector3.right));
18     animator.SetTrigger("Right");
19 }
```

Listing 3.2: Funkcje ruchu w skrypcie *PlayerMovement.cs*

Na Listingu 3.2 zostały przedstawione funkcje odpowiadające za ruch postaci w czterech kierunkach. Każda z tych funkcji wywołuje kurutynę *Move()* z argumentem typu **Vector3**, który odpowiada za kierunek w trzech osiach. Kurutyna *Move()* w zależności od podanego kierunku, porusza obiektem gracza. Skrypt odpowiadający za ruch przeciwnika wygląda podobnie.

```
1 public class EnemyStats : MonoBehaviour {
2
3     [SerializeField] private UIManager uiManager;
4     [SerializeField] private int hp = 100;
5     [SerializeField] private int maxHp = 100;
6     [SerializeField] private int mana = 200;
7     [SerializeField] private int maxMana = 200;
8     [SerializeField] private int regenHp = 5;
9     [SerializeField] private int regenMana = 15;
10
11    public void SetHP(int hp) { this.hp = hp; }
12    public void SetMana(int mana) { this.mana = mana;
          }
13    public int GetHP() { return hp; }
14    public int GetMana() { return mana; }
15    public void RegenStats() {
16        hp += regenHp;
17        mana += regenMana;
18
19        if(hp > maxHp) hp = maxHp;
20        if(mana > maxMana) mana = maxMana;
21
22        uiManager.SetValueToSlider(hp, "ENEMYHP");
23        uiManager.SetValueToSlider(mana, "ENEMYMANA");
24    }
25
26 }
```

Listing 3.3: Klasa *EnemyStats.cs*

Na Listingu 3.3 została pokazana klasa *EnemyStats.cs*, która odpowiada za implementację zasobów przeciwnika. W klasie zostało utworzonych sześć zmiennych całkowitych, **hp**, **maxHp**, **regenHp** odpowiadających za zasób zdrowia przeciwnika oraz **mana**, **maxMana** i **regenMana** odpowiadających za zasób many. Do tego istnieje referencja do klasy UI Manager. Wartości te zostały ustalone tak, jak było to przedstawiane w poprzednim rozdziale. Wartość punktów życia przeciwnika jest ustawiona na 100, wartość maksymalna na 100 oraz regeneracja co turę jest ustawiona na 5. Wartość punktów many przeciwnika jest ustawiona na 200, wartość maksymalna na 200 oraz regeneracja co turę na 15. Poniżej zaimplementowano proste akcesory oraz mutatory do zmiennych życia i many przeciwnika, a ostatnią zaimplementowaną funkcją jest funkcja *RegenStats()*, która odpowiada za regenerację zasobów wraz z początkiem każdej tury. Referencja do klasy UI Manager jest potrzebna, aby na bieżąco ustawiać suwaki życia i many w interfejsie użytkownika podczas gry, co zostanie szerzej omówione w podrozdziale dotyczącym interfejsu gry.

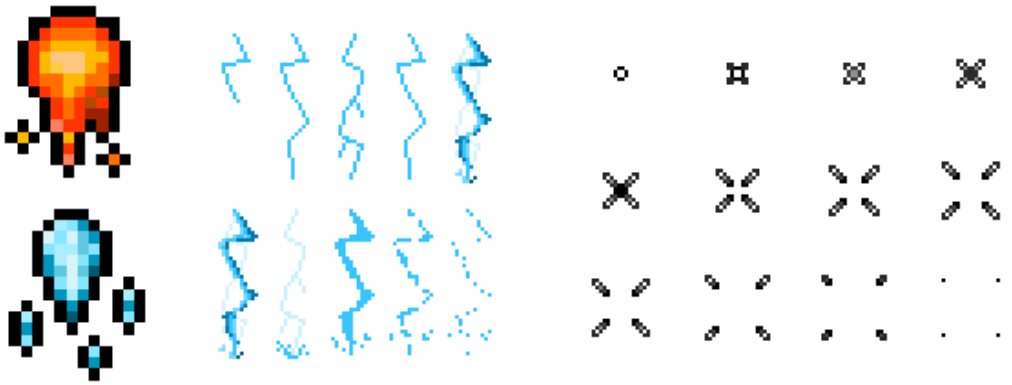
3.5. Ataki, animacje i akcje gracza

Ataków dostępnych dla gracza i przeciwnika jest cztery. Są to Kula ognia, Pocisk lodu, Błyskawica oraz Uderzenie kijem. W tym podrozdziale przedstawione zostanie w jaki sposób zostały stworzone animacje oraz w jaki sposób zostały zaprogramowane ataki i akcje gracza.

Animacje w projekcie można podzielić na dwa rodzaje. Pierwszym rodzajem są animacje ruchu postaci. W tym przypadku możemy wyszczególnić sześć potrzebnych animacji. Są to: animacja postaci będącej w spoczynku, animacja postaci poruszającej się w każdym z czterech kierunków oraz animacja śmierci postaci. Każda z tych animacji wykorzystuje tzw. arkusz sprite'ów *(ang. spritesheet)*, który posiada kilka statycznych grafik wyświetlanych jedna po drugiej bardzo szybko, sprawiając wrażenie animacji. Na Rysunku 3.12 przedstawiony został arkusz sprite'ów dla ruchu przeciwnika.

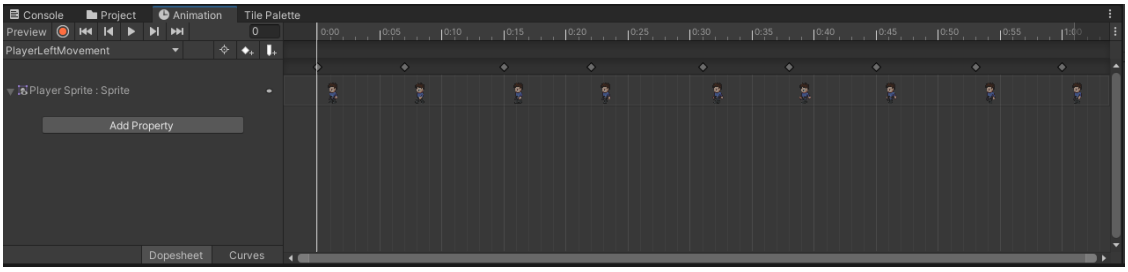
Rysunek 3.12: Arkusz sprite'ów dla ruchu przeciwnika w czterech kierunkach używany do animowania postaci

Drugi rodzaj animacji to animacje ataków postaci. Są one realizowane dwojako. Animacja ataku Kula ognia oraz ataku Pocisk lodu jest animowana w postaci pojedynczej grafiki, która pojawia się na planszy gry i porusza się w stronę celu, sprawiając wrażenie „rzucania czaru". Animacja ataku Błyskawica oraz Uderzenie Kijem jest realizowana tak samo jak animacje ruchu postaci - poprzez odpowiedni spritesheet. Animacje te są automatycznie odtwarzane w miejscu docelowym. Arkusze sprite'ów używane do ruchu postaci zostały wykonane za pomocą darmowego generatora postaci LPC [7], natomiast animacje ataków zostały wykonane w całości przeze mnie i zostały one przedstawione w Rysunku 3.13.

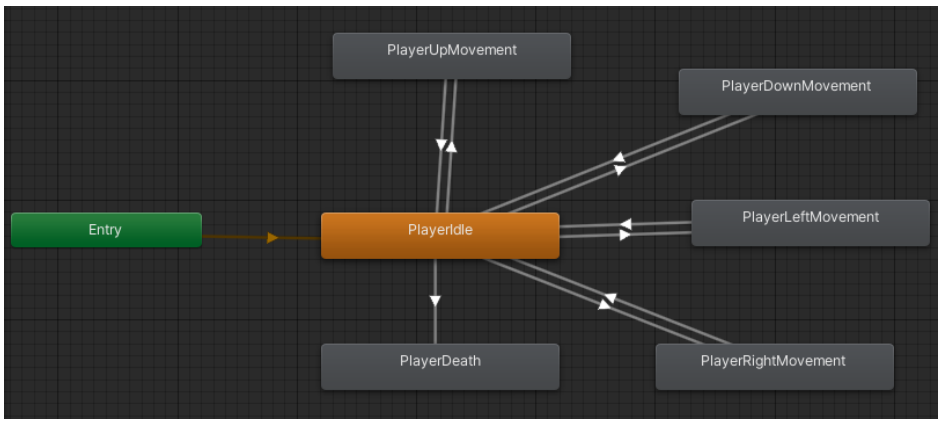


Rysunek 3.13: Grafiki animacji ataków – od góry po lewej: Kula ognia, Pocisk lodu, Błyskawica, Uderzenie Kijem

W Unity animacje realizuje się za pomocą dwóch okienek, *Animation* (przedstawionym na Rysunku 3.14), *Animator* (przedstawionym na Rysunku 3.15) oraz za pomocą skryptów. W okienku *Animation* tworzymy animację na bazie klatek. Tutaj decydujemy o długości animacji, jakie grafiki będą wchodzić w jej skład, można ją odtwarzać i edytować. Okienko *Animator* służy do zarządzania animacjami. Tu decydujemy o całym systemie animacji, która animacja ma być odtwarzana kiedy i pod jakim warunkiem.



Rysunek 3.14: Okienko Animation odpowiedzialne za tworzenie animacji



Rysunek 3.15: Okienko Animator odpowiedzialne za zarządzanie animacjami

Skryptem odpowiadającym za ataki gracza oraz przeciwnika są skrypty *PlayerSpells.cs* oraz *EnemySpells.cs*. Analogicznie jak w przypadku skryptów do poruszania się są one w zasadzie identyczne.

Na samym początku definiowane są prywatne zmienne każdego z ataków, takie jak minimalne obrażenia, maksymalne obrażenia, zasięg, koszt many oraz obiekt animacji. Oprócz tego, istnieje referencja do obiektu celu – w przypadku skryptu gracza celem będzie przeciwnik, w przypadku przeciwnika celem będzie gracz. Dodatkowo zaimplementowane zostały funkcje odpowiedzialne za tworzenie pocisku ataku oraz odtwarzanie animacji.

W skrypcie są zaimplementowane cztery funkcje, a każda z nich odpowiada obsłudze animacji jednego z ataków. Jako że atak Kula ognia oraz Pocisk lodu są animowane w szczególny sposób, w tych funkcjach zostały zaprogramowane specjalne obliczenia pozycji i rotacji pocisku w świecie gry, aby w każdym przypadku animacja odtwarzała się poprawnie. Ataki Błyskawica oraz Uderzenie kijem są zanimowane za pomocą arkusza sprite'ów, zatem jedyną funkcjonalnością metod obsługujących animację tych ataków jest stworzenie obiektu animacji, włączenie animacji oraz zniszczenie obiektu animacji. Jako, że klasy *PlayerSpells.cs* oraz *EnemySpells.cs* pracują z animacjami, dodatkowo zostały wykorzystane kurutyny.

Ostatnim skryptem, który zostanie omówiony w tym podrozdziale, będzie skrypt odpowiadający możliwym akcjom gracza. Posiadając skrypt od poruszania się oraz skrypt od atakowania, można połączyć wreszcie te dwie funkcjonalności w jedną całość. Będzie za to odpowiadał skrypt *PlayerAction.cs*

Skrypt *PlayerAction.cs* posiada łącznie 8 oddzielnych funkcji, które odpowiadają jednej z ośmiu możliwych akcji do wykonania. Składają się na to cztery możliwe ataki oraz ruch w czterech możliwych kierunkach. Tutaj za przykład posłuży funkcja poruszania się do góry, przedstawiona na Listingu 3.4 oraz funkcja ataku Pocisk lodu, przedstawiona na Listingu 3.5

```
1 public void MovingUp() {
2     if(gameManager.GetTurn() == GameManager.TURN.
         PLAYER) {
3         playerMovement.MoveUp();
4         gameManager.AddLogToConsole("Gracz poruszył si
             ę do góry");
5         gameManager.SetTurnTo("ENEMY");
6     }
7 }
```

Listing 3.4: Funkcja *MovingUp()* klasy *PlayerAction.cs*

```
1 public void Frostbolt() {
2     if(gameManager.GetTurn() == GameManager.TURN.
         PLAYER) {
3         playerSpells.Frostbolt();
4         randomDmg = Random.Range(playerSpells.
             GetFrostboltDmgMin(), playerSpells.
             GetFrostboltDmgMax()+1);
5         gameManager.DealDamageTo(randomDmg, "ENEMY");
6         gameManager.SubtractMana(playerSpells.
             GetFrostboltManaCost(), "PLAYER");
7         gameManager.AddLogToConsole("Gracz zaatakował
             pociskiem lodu zadając " + randomDmg + "
             obrażeń");
8         gameManager.SetTurnTo("ENEMY");
9     }
10 }
```

Listing 3.5: Funkcja *Frostbolt()* klasy *PlayerAction.cs*

Funkcja *MovingUp()* odpowiada za ruch gracza w górę. Najpierw sprawdzany jest warunek czy obecną turą w rozgrywce jest tura gracza, aby uniemożliwić graczowi poruszanie się nie w swojej turze. Jeżeli warunek jest spełniony, wywoływana jest funkcja *MoveUp()* z klasy *PlayerMovement.cs*. Następnie dodawany jest wpis do konsoli i następuje zmiana tury.

Funkcja *Frostbolt()* odpowiada za atak Pocisk lodu. Analogicznie, najpierw sprawdzany jest warunek obecnej tury. Jeśli warunek jest spełniony, wywoływana jest funkcja *Frostbolt()* z klasy *PlayerSpells.cs* odpowiadająca za animację. Następnie losowane są obrażenia z zakresu minimum-maksimum odpowiadające atakowi Pocisk lodu. Po wylosowaniu obrażeń, następne dwie linijki wywołują funkcje od zadawania obrażeń oraz odbierania zasobu many zaimplementowane w menedżerze gry. Na samym końcu dodawany jest wpis do konsoli i następuje zamiana tury.

Aby te wszystkie funkcje mogły być realizowane w czasie rzeczywistym, muszą zostać obsłużone w systemowej funkcji *Update()*, która wywołuje się co klatkę gry. Niestety, Unity nie posiada domyślnie wgranego systemu sprawdzającego naciśnięte klawisze w trakcie rozgrywki, zatem jeśli gra ma zawierać sterowanie klawiaturą, trzeba napisać dość brzydko wyglądające drzewko *if-else* dla każdego przypadku. Oprócz sterowania klawiaturą zostanie zaimplementowane również sterowanie myszką, o którym szerzej będzie w podrozdziale odpowiadającym interfejsowi użytkownika. W Tabeli 3.1 zostanie przedstawione jak wygląda sterowanie klawiaturą w grze.

Klawisz	Alternatywny klawisz	Akcja
W	Strzałka do góry	Ruch do góry
S	Strzałka do dołu	Ruch do dołu
A	Strzałka w lewo	Ruch w lewo
D	Strzałka w prawo	Ruch w prawo
1	–	Kula ognia
2	–	Pocisk lodu
3	–	Błyskawica
4	–	Uderzenie kijem

Tabela 3.1: Obsługa klawiatury w grze

Schemat działania w funkcji *Update()* za każdym razem wygląda identycznie. Najpierw sprawdzany jest warunek naciśnięcia któregoś z 12 klawiszy, a następnie wywoływane są oddzielne funkcje odpowiadające każdej akcji. Jedyną różnicą są klawisze odpowiadające za akcje ataku, gdyż zanim zostanie wywołana funkcja ataku, sprawdzane są dodatkowo dwa warunki. Pierwszym z nich jest warunek posiadania odpowiedniej ilości zasobu punktów many, a drugim z nich jest warunek posiadania odpowiedniego zasięgu.

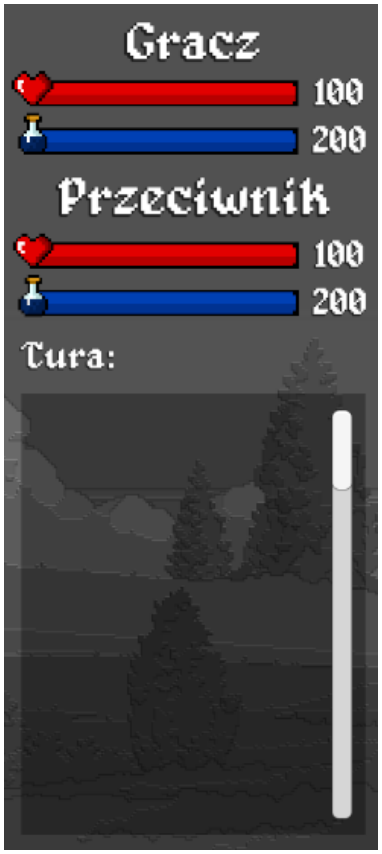
3.6. Interfejs użytkownika w trakcie rozgrywki

Interfejs użytkownika został zrealizowany w postaci dwóch paneli widocznych cały czas w trakcie rozgrywki. Panel boczny, po stronie prawej, wyświetla zasoby gracza oraz przeciwnika, obecną turę oraz konsolę, w której rejestrowana jest każda tura. Panel dolny służy do obsługi akcji gracza za pomocą myszki. Znajdują się tam przyciski wszystkich czterech ataków oraz cztery przyciski służące do poruszania się.

Unity posiada odpowiednie narzędzia do tworzenia i wyświetlania interfejsu użytkownika, z angielskiego nazwanego UI *(User Interface)*. Aby móc wyświetlać elementy interfejsu takie jak przyciski, obrazki, okienka, panele czy tekst, najpierw trzeba utworzyć płótno *(ang. canvas)*.

Razem z utworzeniem nowego płótna, w hierarchii automatycznie tworzy się również obiekt *EventSystem*, odpowiedzialny za obsługiwanie takich zdarzeń jak na przykład naciśnięcie przycisku. Wszelkie pozostałe elementy UI, które będą tworzone muszą być tworzone jako dzieci obiektu Canvas.

Panel boczny, który znajduje po prawej stronie został utworzony poprzez element *Panel* z menu kontekstowego UI. Jego tłem jest grafika na darmowej licencji z paczki assetów Free Nature Background Pixel Art autorstwa twórców strony craftpix.net [8], odpowiednio zmodyfikowana przeze mnie w programie graficznym GIMP. Użytą czcionką w interfejsie użytkownika (oraz we wszystkich innych oknach gry) jest darmowa czcionka Alagard autorstwa Hewett Tsoi [9].



Rysunek 3.16: Wygląd panelu bocznego

Panel boczny (przedstawiony na Rysunku 3.16) składa się z wcześniej wymienionych elementów, takich jak zasoby gracza, zasoby przeciwnika, informacja o obecnej turze oraz z konsoli, wyświetlającej cały zapis rozgrywki. Grafiki użyte do wyświetlania pasków punktów życia oraz punktów many zostały stworzone w całości przeze mnie. Czerwony pasek przedstawia punkty życia, niebieski pasek przedstawia punkty many. Zaciemniony obszar wraz z paskiem przewijania służy do wyświetlania konsoli.

Panel dolny (przedstawiony na Rysunku 3.17) składa się wyłącznie z przycisków do obsługi gracza. Znajdują się tam cztery przyciski symbolizujące atak gracza (od lewej: Kula ognia, Pocisk lodu, Błyskawica, Uderzenie kijem) oraz cztery przyciski ułożone na kształt klawiszy strzałek na klawiaturze, symbolizujące przyciski ruchu. Sam panel nie posiada grafiki tła, jest przezroczysty. Pozostałe grafiki zostały użyte na darmowej licencji. Grafika przycisku Kula ognia została wykorzystana z paczki assetów Basic RPG Icons autorstwa PONETI [10]. Grafika przycisku Pocisk Lodu pochodzi z paczki assetów FREE - RPG Fantasy Spell Icons autorstwa użytkownika Blink [11]. Grafika przycisku Błyskawica ma źródło w paczce assetów Awesome Fantasy Skill Icon pack autorstwa ji-hyeeeee [12], a grafika przycisku Uderzenie Kijem to grafika z paczki Free 50 Aeromancer Skills użytkownika free-game-assets [13]. Grafika klawiszy strzałek pochodzi z paczki BlatKeys autorstwa BlatFan [14].

Rysunek 3.17: Wygląd panelu dolnego

Aby obsłużyć cały interfejs różnymi „ułatwiaczami", tzw. QOL *(ang. Quality of Life)* i aby wyglądał on zgrabnie, został utworzony skrypt *UIManager.cs*. Analogicznie jak w przypadku menedżera gry, tu również został utworzony pusty obiekt w świecie gry, którego zadaniem jest tylko wywoływanie skryptu *UIManager.cs* w czasie rzeczywistym. W kontekście interfejsu podczas rozgrywki, zawiera on trzy pomocnicze funkcje, które zostaną pokrótce omówione. Większą rolę skrypt *UIManager.cs* odgrywa w obsłudze początkowych i końcowych ekranów gry, o których będzie szerzej w ostatnim podrozdziale.

1. *SetValueToSlider()* – funkcja odpowiadająca za obsługę suwaków punktów życia oraz many gracza i przeciwnika w panelu bocznym. Pobiera ona wartość oraz nazwę suwaka i odpowiednio go dostosowuje. Jest ona wykorzystywana zawsze tam, gdzie następuje zmiana wartości zasobów postaci i dla przykładu było widać jej wywołanie w podrozdziale 3.3 przy omawianiu funkcji *RegenStats()*, która odpowiada za regenerację zasobów.

2. *AddLogToConsole()* – funkcja odpowiadająca za dodawanie tekstu do konsoli w panelu bocznym. Każda akcja wykonana przez gracza lub przeciwnika jest rejestrowana automatycznie w konsoli, za co odpowiada ta funkcja.

3. *SetActiveButtons()* – funkcja odpowiadająca za włączanie i wyłączanie klawiszy ataku w panelu dolnym w zależności od tego, czy atak jest możliwy do wykonania czy nie.

Sama klasa *UIManager.cs* posiada również mnóstwo referencji do wielu zmiennych z różnych funkcji oraz obiektów ze świata gry, aby móc je odpowiednio obsłużyć. Oprócz funkcji *UIManager.cs* istnieje jeszcze poboczna klasa *TooltipDisplayer.cs*, która odpowiada tylko i wyłącznie za wyświetlenie dymka podpowiedzi przy najechaniu na któryś atak, co zostało pokazane na Rysunku 3.18.

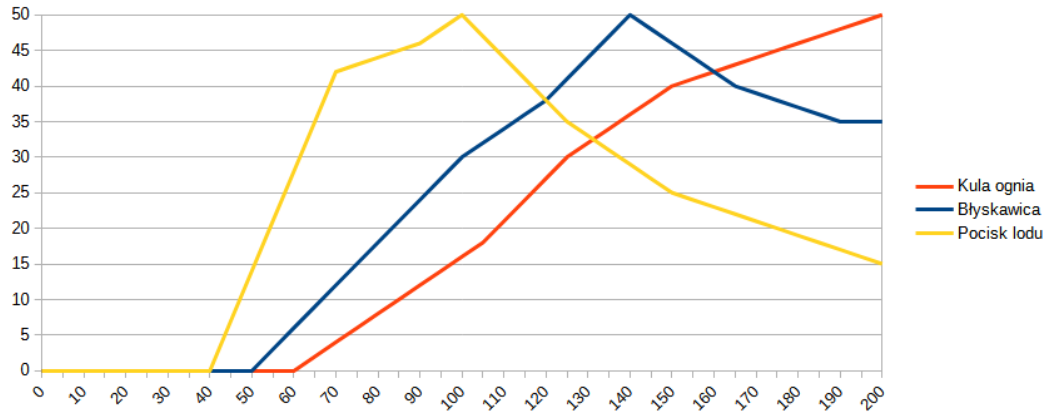
Rysunek 3.18: Przykład zastosowania klasy *TooltipDisplayer.cs*

3.7. Sztuczna inteligencja

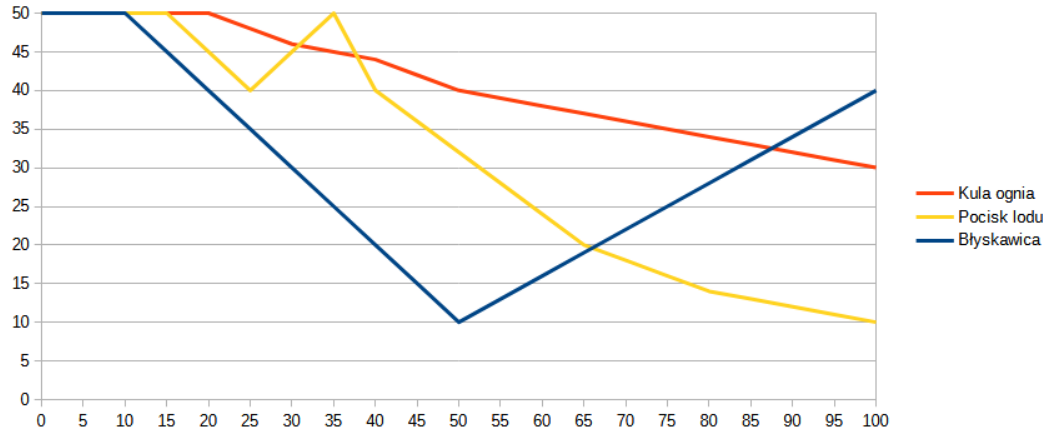
W poprzednim rozdziale zostało omówione, w jaki sposób będzie przebiegać implementacja sztucznej inteligencji przeciwnika. Przeciwnik posiada dwa stany, stan patrolowania oraz stan atakowania. W stanie patrolowania porusza się on w jednym z czterech kierunków losowo. W stanie atakowania dobór ataku jest rozstrzygany na podstawie logiki rozmytej.

Za całość sztucznej inteligencji odpowiadają dwa skrypty, *EnemyFuzzyLogic.cs* oraz *EnemyAI.cs*. Pierwszy z nich odpowiada za ewaluację ataku na podstawie danych wejściowych, tj. punktów życia gracza oraz punktów many przeciwnika. Drugi skrypt odpowiedzialny jest za obsłużenie stanów przeciwnika i wywoływań odpowiednich funkcji ruchu bądź ataku z innych skryptów, bardzo podobnie jak robił to skrypt *PlayerAction.cs*.

Aby przenieść logikę rozmytą do silnika gry Unity, zostaną zastosowane tzw. *Animation Curves*. Jest to narzędzie Unity, które pierwotnie miało zastosowanie do tworzenia wykresów animacji w czasie, natomiast w trakcie korzystania z nich developerzy i game designerzy zauważyli potencjał do wykorzystania ich w innych dziedzinach tworzenia gier. To, co jest przydatne w krzywych animacji to fakt, że za ich pomocą można narysować pewien wykres, a następnie wywołać na nim funkcję *Evaluate()*, która zwróci nam wartość dla zadanego argumentu. Dzięki temu nie ma potrzeby tworzyć wielkich tablic, aby przechowywać wartości liczbowych dla każdej możliwej sytuacji, która może zostać napotkana. Zamiast tego, można narysować wykres a następnie wywoływać jedną funkcję aby wydobyć z niego wartość. W poprzednim rozdziale, gdzie została omówiona szczegółowo logika rozmyta, zostały przedstawione dwa wykresy łączące wartości liczbowe dla poszczególnych ataków. Jeden wykres przedstawiał wartości liczbowe w zależności od punktów życia gracza, a drugi wykres przedstawiał wartości liczbowe w zależności od punktów many przeciwnika. Na Rysunku 3.19 oraz 3.20 zostały one przypomniane.



Rysunek 3.19: Wykres ewaluacji trzech ataków przeciwnika na podstawie jego punktów many

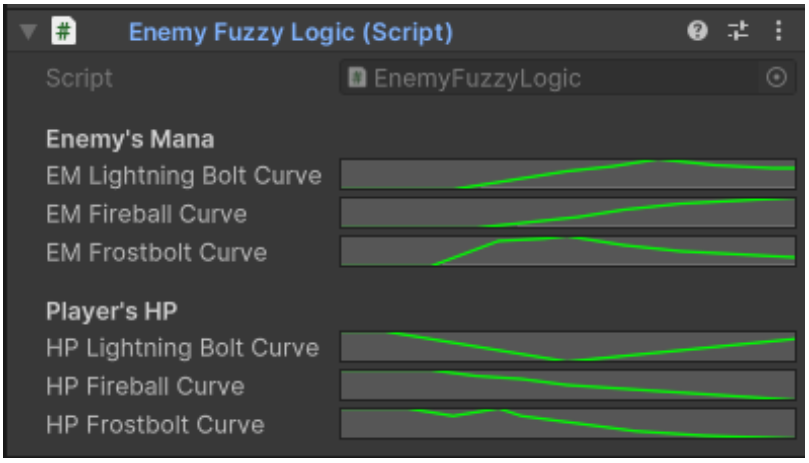


Rysunek 3.20: Wykres ewaluacji trzech ataków przeciwnika na podstawie punktów życia gracza

Zostaną one przeniesione do silnika gry Unity, natomiast wcześniej będą one rozbite dla każdego pojedynczego ataku, dając całościowo sześć możliwych wykresów. Zatem w skrypcie *EnemyFuzzyLogic.cs* zostało utworzonych sześć prywatnych zmiennych typu AnimationCurve z dodatkowym atrybutem [SerializeField], aby można było je edytować z poziomu inspektora, co zostało przedstawione na Listingu 3.6. Na rysunku 3.21 został pokazany widok edycji utworzonych zmiennych z poziomu inspektora.

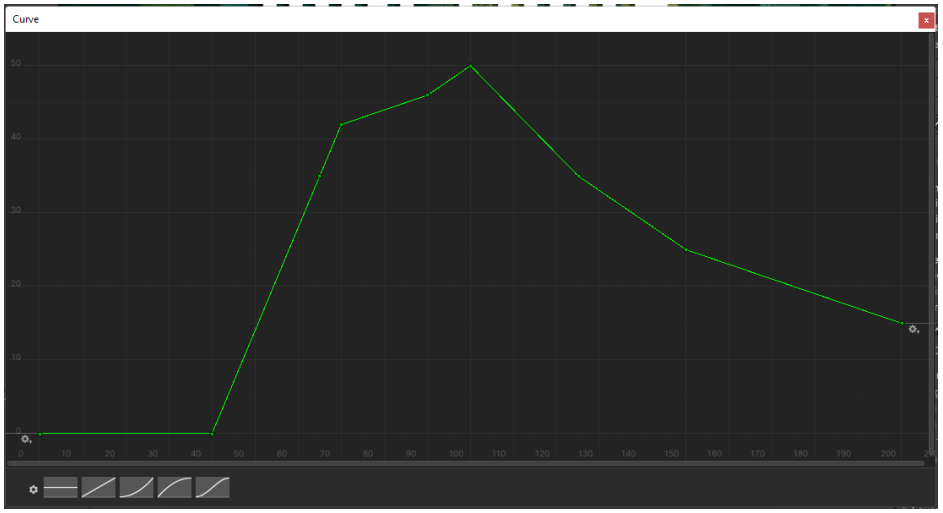
```
1     [Header("Enemy's Mana")]
2     [SerializeField] private AnimationCurve
        EMLightningBoltCurve;
3     [SerializeField] private AnimationCurve
        EMFireballCurve;
4     [SerializeField] private AnimationCurve
        EMFrostboltCurve;
5
6     [Header("Player's HP")]
7     [SerializeField] private AnimationCurve
        HPLightningBoltCurve;
8     [SerializeField] private AnimationCurve
        HPFireballCurve;
9     [SerializeField] private AnimationCurve
        HPFrostboltCurve;
```

Listing 3.6: Zmienne AnimationCurves w kodzie skryptu *EnemyFuzzyLogic.cs*

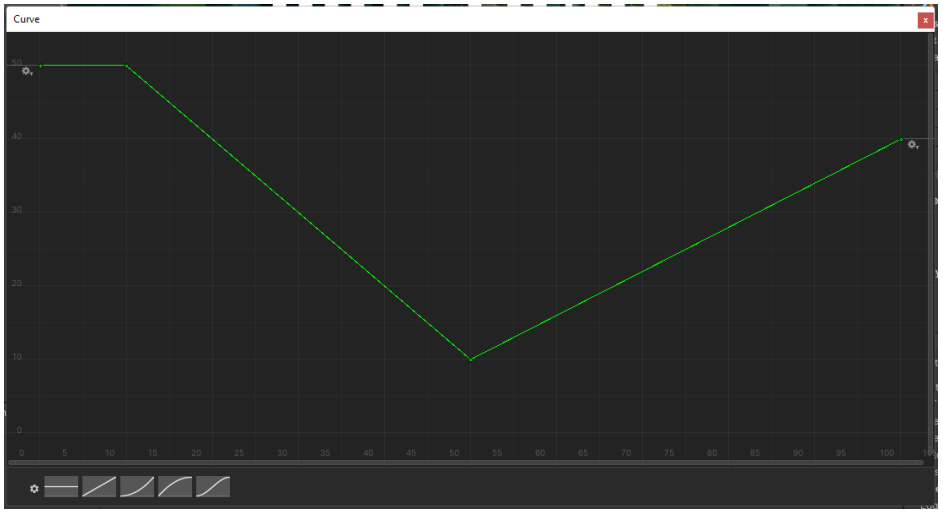


Rysunek 3.21: Wygląd utworzonych krzywych w inspektorze, skąd można je edytować

Tworzenie krzywych w edytorze Unity jest trywialne i wymaga tylko kilku kliknięć, aby utworzyć oczekiwany wykres. Na Rysunku 3.22 oraz 3.23 zostały przedstawione dwie krzywe i ich wygląd w edytorze, odpowiadające wartościom dla ataku Pocisk Lodu w zależności od punktów many przeciwnika oraz wartościom dla ataku Błyskawica w zależności od punktów życia gracza.



Rysunek 3.22: Wygląd krzywej EM Frostbolt, która odpowiada za wartości ataku Pocisk Lodu na podstawie punktów many przeciwnika



Rysunek 3.23: Wygląd krzywej HP Lightning Bolt, która odpowiada za wartości ataku Błyskawica na podstawie punktów życia gracza

Reszta kodu w skrypcie *EnemyFuzzyLogic.cs* to szereg prywatnych zmiennych pomocniczych oraz funkcja *EvaluateSpell()*, która dokonuje wyboru ataku oraz zwraca go na podstawie danych wejściowych – punktów życia gracza oraz punktów many przeciwnika. Funkcja ta została przedstawiona w Listingu 3.7

```
1 public string EvaluateSpell(int enemyMana, int
    playerHP) {
2
3     if(enemyMana < 40) {
4         spellString = "MAGICSTAFF";
5     } else {
6         EMLightningBoltCurveValue =
              EMLightningBoltCurve.Evaluate(enemyMana);
7         EMFireballCurveValue = EMFireballCurve.
              Evaluate(enemyMana);
8         EMFrostboltCurveValue = EMFrostboltCurve.
              Evaluate(enemyMana);
9         HPLightningBoltCurveValue =
              HPLightningBoltCurve.Evaluate(playerHP);
10        HPFireballCurveValue = HPFireballCurve.
              Evaluate(playerHP);
11        HPFrostboltCurveValue = HPFrostboltCurve.
              Evaluate(playerHP);
12
13        lightningBoltFinalValue =
              EMLightningBoltCurveValue +
              HPLightningBoltCurveValue;
14        fireballFinalValue = EMFireballCurveValue +
              HPFireballCurveValue;
```

```
15          frostboltFinalValue = EMFrostboltCurveValue +
               HPFrostboltCurveValue;
16
17          setA = lightningBoltFinalValue;
18          setB = lightningBoltFinalValue +
               fireballFinalValue;
19          setC = lightningBoltFinalValue +
               fireballFinalValue + frostboltFinalValue;
20
21          chosenSpell = Random.Range(0, setC);
22
23          if(chosenSpell >= 0 && chosenSpell < setA) {
24              spellString = "LIGHTNING";
25          } else if(chosenSpell >= setA && chosenSpell <
                setB) {
26              spellString = "FIRE";
27          } else {
28              spellString = "FROST";
29          }
30      }
31
32      return spellString;
33  }
```

Listing 3.7: Funkcja *EvaluateSpell()* klasy *EnemyFuzzyLogic.cs*

Na samym starcie jest sprawdzany warunek, czy przeciwnik posiada mniej niż 40 punktów many. Jeśli tak, wybranym atakiem jest atak Uderzenie kijem o kodowej nazwie „MAGICSTAFF". Jeżeli przeciwnik posiada więcej niż 40 punktów many, to znaczy że jest w stanie wykonać co najmniej jeden z trzech pozostałych ataków. Linijki 6-11 odpowiadają za ewaluację każdej z sześciu krzywych na postawie argumentów przekazanych w funkcji i zapisanie tych wartości do sześciu pomocniczych zmiennych. Następnie wartości dla każdego ataku są sumowane oraz tworzone są trzy zbiory o liczebności każdego z ataków. Na samym końcu następuje losowanie liczby z pomiędzy wartości 0 a sumy wszystkich zbiorów. Wylosowana liczba musi należeć do któregoś ze zbioru, zatem w każdym z trzech przypadków jest zwracany odpowiednio wybrany atak. Tak wybrany atak o odpowiedniej nazwie kodowej jest następnie zwracany w postaci łańcucha znaków. Funkcja *EvaluateSpell()* jest następnie wykorzystywana w skrypcie *EnemyAI.cs*.

Skrypt *EnemyAI.cs* odpowiada ostatecznie za całość wykonywania akcji przeciwnika. Umieszczona w nim została obsługa dwóch stanów zachowania, ruchu postaci i ataku. Klasa ta, podobnie jak klasa *PlayerAction,cs*, posiada referencje do poprzednich skryptów, które obsługują ruch przeciwnika, jego zasoby, możliwe ataki oraz logikę rozmytą. Dodatkowo została utworzona referencja do obiektu gracza, który jest celem ataków przeciwnika, jego statystyk, aby wydobyć wartość jego punktów życia, potrzebną do ewaluacji ataku oraz referencja do menedżera gry, aby obsłużyć przekazanie tury.

W klasie najpierw został zaimplementowany enumerator z każdym możliwym stanem przeciwnika, oraz prywatne pole przechowujące aktualny stan, analogicznie jak w przypadku menedżera gry. Oprócz tego, klasa zawiera kilka pomocniczych funkcji przy realizowaniu każdego stanu. W momencie, gdy przeciwnik na podstawie danych wejściowych wybierze atak, którego nie jest w stanie wykonać, ze względu na zbyt dużą odległość od gracza, akcją w turze przeciwnika powinien być ruch w stronę gracza, aby tę odległość zminimalizować. Odpowiada za to pomocnicza funkcja *MoveTowardsPlayer()*. Analizuje ona pozycję przeciwnika względem gracza i na tej podstawie wykonuje ruch w kierunku niego. Drugą pomocniczą funkcją jest funkcja *CheckStateConditions()*, która odpowiada za sprawdzenie warunku przejścia pomiędzy stanem „Patroluj", a stanem „Atakuj", czyli warunek odległości równej bądź mniejszej niż 6 kratek pomiędzy graczem a przeciwnikiem.

Każdy stan przeciwnika posiada oddzielną funkcję, która następnie jest wywoływana w zależności od wartości zmiennej **state**. W funkcji *PatrollingState()*, która jest widoczna na Listingu 3.8 zaimplementowany został stan „Patroluj". Zanim przeciwnik wykona ruch, za każdym razem odczekuje 2 sekundy. W tym przypadku została zastosowana sztuczka z odmierzaniem czasu za pomocą zmiennej *Time.deltaTime* zamiast używania kurutyn. Jest to technika mniej zgrabna, ale za to łatwiejsza do zrozumienia. W stanie „Patroluj" przeciwnik losuje ruch w jednym z czterech kierunków. W tym wypadku losuje liczbę z przedziału 0-4, wyłączając liczbę cztery, a następnie została zaimplementowana prosta struktura *switch case*, aby obsłużyć ruch w każdym z czterech możliwych kierunków. Na samym końcu resetowany jest timer zliczający dwie sekundy oraz tura zostaje przekierowywana do gracza.

```
1 private void PatrollingState() {
2
3     timer += Time.deltaTime;
4
5     if (timer > 2) {
6         randomDirection = Random.Range(0, 4);
7
8         switch(randomDirection) {
9             case 0: {
10                enemyMovement.MoveUp();
11                gameManager.AddLogToConsole("
                      Przeciwnik poruszył się do góry");
12                break;
13            }
14            case 1: {
15                enemyMovement.MoveDown();
16                gameManager.AddLogToConsole("
                      Przeciwnik poruszył się do dołu");
17                break;
18            }
19            case 2: {
20                enemyMovement.MoveLeft();
21                gameManager.AddLogToConsole("
                      Przeciwnik poruszył się w lewo");
22                break;
23            }
24            case 3: {
25                enemyMovement.MoveRight();
26                gameManager.AddLogToConsole("
                      Przeciwnik poruszył się w prawo");
27                break;
28            }
29        }
30
31        gameManager.SetTurnTo("PLAYER");
32        timer = 0;
33    }
34 }
```

Listing 3.8: Funkcja *PatrollingState()*

Stan „Atakuj” jest obsłużony w funkcji *ActionState()*, której wycinek został przedstawiony na Listingu 3.9. Podobnie jak i w poprzednim przypadku, najpierw przeciwnik czeka dwie sekundy ze swoim ruchem. Następnie wywołuje funkcję *EvaluateSpell()* z klasy *EnemyFuzzyLogic.cs*. W zależności od tego, jaka nazwa kodowa ataku została zwrócona w funkcji, prosta konstrukcja *switch case* warunkuje który atak ma zostać wykonany. W każdym oddzielnym przypadku, zanim atak zostanie wykonany, sprawdzany jest zasięg czaru i porównywany z odległością od gracza. Tak jak było wspominane wcześniej, jeżeli wylosowany atak nie może dojść do skutku ze względu na zbyt dużą odległość, przeciwnik powinien poruszyć się w stronę gracza, aby tę odległość zmniejszyć. Tutaj więc, jeżeli warunek odległości nie został spełniony, zamiast ataku wywoływana jest omówiona wcześniej funkcja *MoveTowardsPlayer()* i tura jest przekazywana graczowi. Jeśli warunek został spełniony, następuje szereg akcji bardzo podobny do tego, jaki był w przypadku gracza w funkcji *PlayerAction.cs*. Najpierw jest odtwarzana animacja ataku, następnie losowane są obrażenia z zakresu minimum-maksimum dla wybranego ataku, potem obrażenia są zadawane oraz odejmowane są punkty many. Na samym końcu następuje przekazanie tury graczowi.

```
1  private void ActionState() {
2
3    timer += Time.deltaTime;
4
5    if (timer > 2)  {
6
7      spell = enemyFuzzyLogic.EvaluateSpell(enemyStats.
         GetMana(), playerStats.GetHP());
8
9      switch(spell) {
10       case "LIGHTNING": {
11
12         if(Mathf.Abs(transform.position.x - player.
             transform.position.x) <= enemySpells.
             GetLightningBoltRange() && Mathf.Abs(
             transform.position.y - player.transform.
             position.y) <= enemySpells.
             GetLightningBoltRange() && enemyStats.
             GetMana() >= enemySpells.
             GetLightningBoltManaCost()) {
13           enemySpells.LightningBolt();
14           randomDmg = Random.Range(enemySpells.
               GetLightningBoltDmgMin(), enemySpells.
               GetLightningBoltDmgMax()+1);
15           gameManager.DealDamageTo(randomDmg, "PLAYER"
               );
16           gameManager.SubtractMana(enemySpells.
               GetLightningBoltManaCost(), "ENEMY");
```

```
17            gameManager.AddLogToConsole("Przeciwnik
                 zaatakował błyskawicą zadając " +
                 randomDmg + " obrażeń");
18            gameManager.SetTurnTo("PLAYER");
19            timer = 0;
20
21          } else {
22            MoveTowardsPlayer();
23            gameManager.SetTurnTo("PLAYER");
24            timer = 0;
25          }
26
27          break;
28        }
29        case "FIRE": {
30          [...]
31        }
32      }
33    }
```

Listing 3.9: Wycinek funkcji *ActionState()* dla ataku Błyskawica

Wszystkie te funkcje są wykorzystywane ostatecznie w systemowych funkcjach Unity *Start()* oraz *Update()*, aby ich funkcjonalność mogła być realizowana w czasie rzeczywistym, co zostało pokazane na Listingu 3.10.

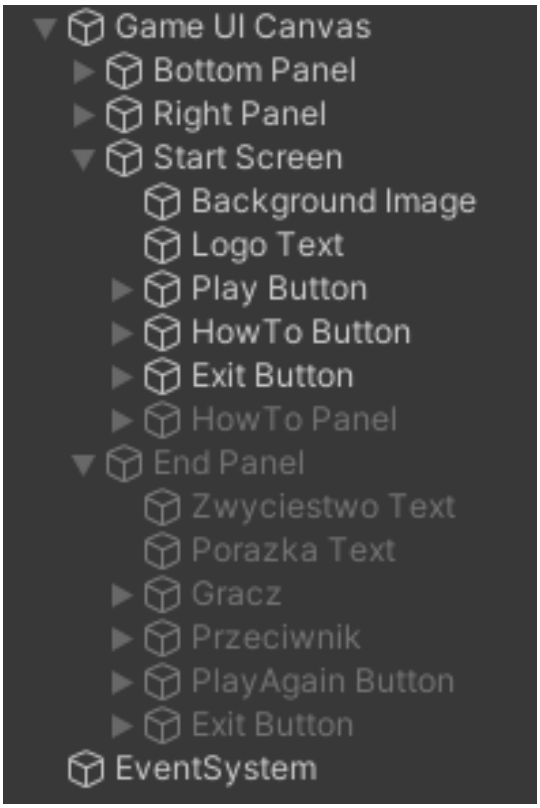
```
1 void Start() {
2     state = STATE.PATROL;
3     timer = 0f;
4 }
5
6 void Update() {
7
8     if (gameManager.GetTurn() == GameManager.TURN.
         ENEMY) {
9
10        CheckStateConditions();
11
12        if (state == STATE.PATROL) {
13            PatrollingState();
14        }
15
16        else { ActionState(); }
17    }
18 }
```

Listing 3.10: Funkcje *Start()* i *Update()*

W funkcji *Start()* ustawiany jest początkowy stan przeciwnika, czyli stan patrolowania. Resetowany jest również timer zliczający sekundy. W funkcji *Update()* sprawdzana jest na samym początku tura rozgrywki, i jeśli obecną turą jest tura przeciwnika, w zależności od wartości zmiennej **state** jest realizowana funkcja *PatrollingState()* lub *ActionState()*. Wcześniej oczywiście jest sprawdzany warunek przejścia.

3.8. Ekrany początkowe i końcowe, menu gry

W ostatnim podrozdziale zostaną omówione ekrany początkowe oraz końcowe gry, to jak wygląda menu i wszystko, co zostało zbudowane wokół samej rozgrywki. Szerzej zostanie również przedstawiona klasa *UIManager.cs*, która poza interfejsem rozgrywki obsługuje również ekrany początkowe i końcowe.

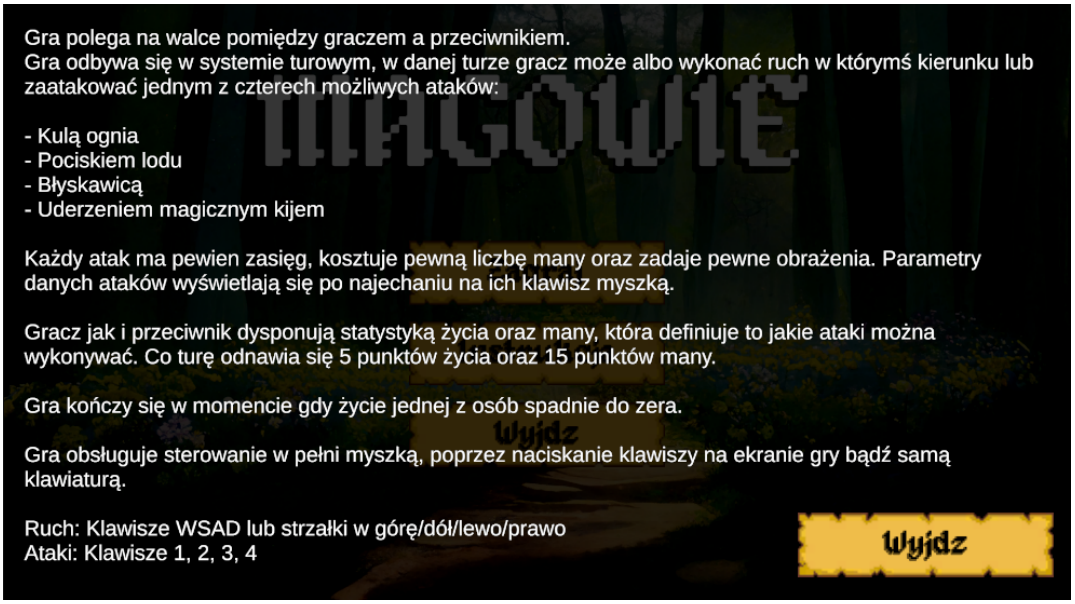


Rysunek 3.24: Hierarchia obiektu Game UI Canvas, odpowiedzialnego za interfejs gry

W hierarchii, widocznej na Rysunku 3.24, zostały utworzone dwa nowe panele o nazwach Start Screen oraz End Panel. Pierwszy z nich odpowiada za ekran początkowy wraz z menu gry, widoczny po włączeniu aplikacji. Drugi pojawia się w momencie śmierci którejś z postaci. Ekran początkowy zawiera nazwę gry, tło na darmowej licencji autorstwa Lornn z paczki assetów Fantasy Forest Backgrounds [15] oraz trzy przyciski – Zagraj, Instrukcja oraz Wyjdź. Tłem dla przycisków jest również grafika na darmowej licencji, tym razem autorstwa Zed Hanok z paczki

Misc HUD and UI Graphics [16]. Przycisk „Zagraj” przenosi gracza do rozgrywki, ustawiając pierwszą turę na turę gracza. Przycisk „Instrukcja” włącza dodatkowy panel o nazwie „HowToPanel”, który domyślnie jest wyłączony (można zauważyć wyszarzony obiekt panelu w hierarchii). Wyświetla on prostą instrukcję gry, tłumaczy zasady rozgrywki oraz sterowanie. Przycisk „Wyjdź” wychodzi z aplikacji. Ekran początkowy został zaprezentowany na Rysunku 3.25, a ekran instrukcji na Rysunku 3.26.

Rysunek 3.25: Ekran początkowy gry



Rysunek 3.26: Panel instrukcji gry

Ekran końcowy gry wyświetla napis „Zwycięstwo” lub „Porażka”, w zależności od końcowego rozstrzygnięcia gry. Oprócz tego, panel ten pokazuje zasoby gracza i przeciwnika w chwili śmierci oraz dwa przyciski: Zagraj jeszcze raz lub Wyjdź. Przycisk „Zagraj jeszcze raz” resetuje całą aplikację do stanu początkowego, a przycisk „Wyjdź” wychodzi z niej. Ekran zwycięstwa widoczny jest na Rysunku 3.27, a ekran porażki na Rysunku 3.28.

Rysunek 3.27: Ekran zwycięstwa

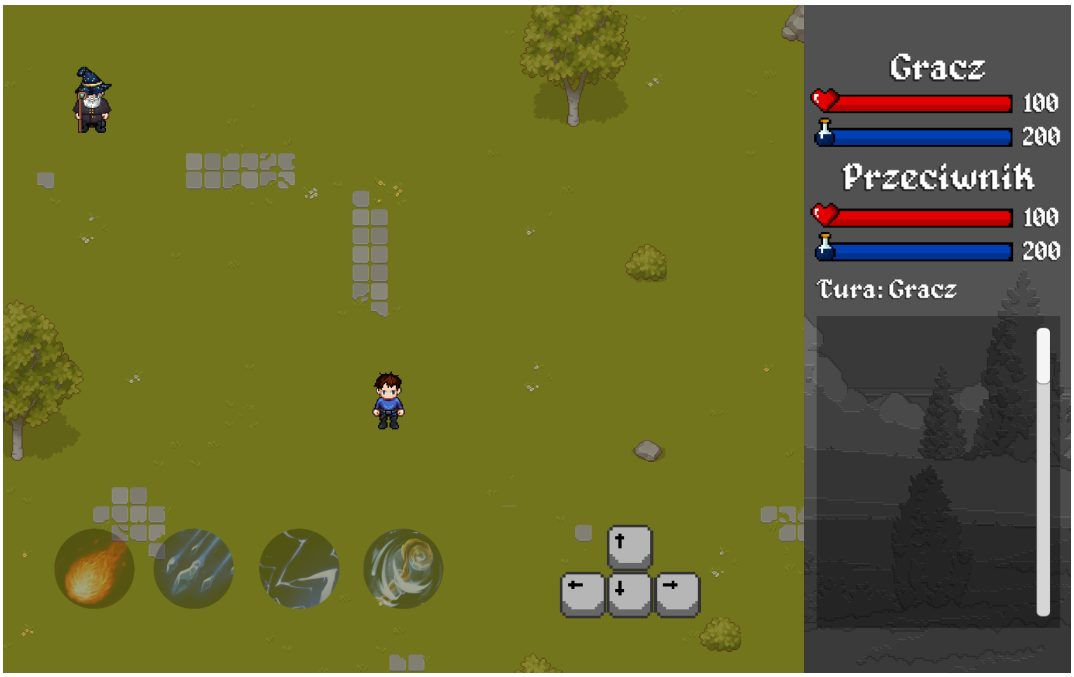
Rysunek 3.28: Ekran porażki

Rozdział 4

Obsługa programu i przebieg rozgrywki

W ostatnim rozdziale pracy zostanie przedstawiona obsługa programu oraz przykładowa rozgrywka wraz z omówieniem poszczególnych ruchów przeciwnika.

Na samym starcie po włączeniu aplikacji użytkownika wita ekran początkowy, który był pokazywany przy okazji poprzedniego rozdziału. Użytkownik ma do wyboru trzy opcje – Zagraj, Instrukcja oraz Wyjdź. Przycisk „Zagraj" przenosi użytkownika od razu do rozgrywki. Przycisk „Instrukcja" wyświetla dodatkowy panel, w którym użytkownik może poznać zasady gry oraz sterowanie. Przycisk „Wyjdź" wychodzi z aplikacji.



Rysunek 4.1: Ekran rozgrywki

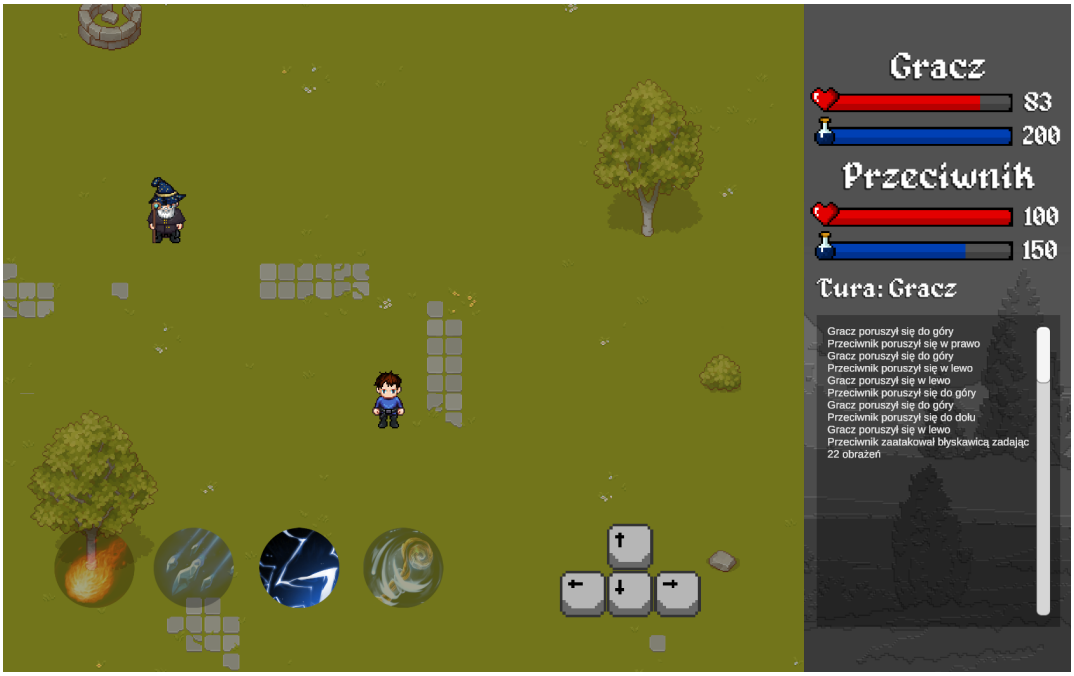
Kliknięcie przycisku „Start” zaczyna faktyczną rozgrywkę. Ekran rozgrywki został przedstawiony na Rysunku 4.1. Na środku znajduje się gracz, którym użytkownik steruje. Razem z postacią gracza porusza się również kamera. W lewym górnym rogu znajduje się przeciwnik. W panelu po prawej znajdują się zasoby gracza oraz przeciwnika – ich punkty życia oraz punkty many. Dodatkowo widać obecną turę oraz konsolę, która będzie zapisywać i wyświetlać każdy ruch gracza oraz przeciwnika. Na dole mamy interfejs sterowania. Widać na nim cztery możliwe do wykonania ataki oraz klawisze strzałek obsługujące ruch. Poza sterowaniem myszką, zostało zaimplementowane również sterowanie klawiaturą, której opis znajduje się w Tabeli 3.1 w poprzednim rozdziale.

Na samym starcie użytkownik może wykonać tylko akcję ruchu, gdyż odległość od przeciwnika wynosi więcej, niż zasięg jakiegokolwiek ataku. Przeciwnik na samym starcie jest w stanie „Patroluj”, co oznacza, że porusza się w losowym kierunku, patrolując obszar wokół siebie w odległości 6 kratek. Po wykonaniu kilku tur, konsola z zapisem gry prezentuje się jak na Listingu 4.1.

```
1 Gracz poruszył się do góry
2 Przeciwnik poruszył się w prawo
3 Gracz poruszył się do góry
4 Przeciwnik poruszył się w lewo
5 Gracz poruszył się w lewo
6 Przeciwnik poruszył się do góry
7 Gracz poruszył się do góry
8 Przeciwnik poruszył się do dołu
9 Gracz poruszył się w lewo
10 Przeciwnik zaatakował błyskawicą zadając 22 obrażeń
```

Listing 4.1: Zapis konsoli po wykonaniu kilku pierwszych tur

Przez pierwsze kilka tur gracz oraz przeciwnik wykonywali akcję ruchu. W momencie, w którym gracz zbliżył się na odległość 6 kratek przeciwnik przeszedł w stan „Atakuj”. Podczas pierwszego ataku przeciwnika przeciwnik jak i gracz posiadają maksymalną ilość zasobów, tzn. gracz posiada 100 punktów życia, a przeciwnik posiada 200 punktów many. W takich warunkach ewaluacja ataków wygląda następująco: **75** punktów dla ataku Błyskawica, **80** punktów dla ataku Kula ognia oraz **25** punktów dla ataku Pocisk lodu. Z tego wynika, że najbardziej prawdopodobnym atakiem do wyboru będzie atak Błyskawica bądź atak Kula ognia. Przeciwnik zdecydował się na wybór pierwszego ataku, zadając 22 obrażeń. Na Rysunku 4.2 pokazany został ekran rozgrywki po wykonaniu kilku pierwszych tur.



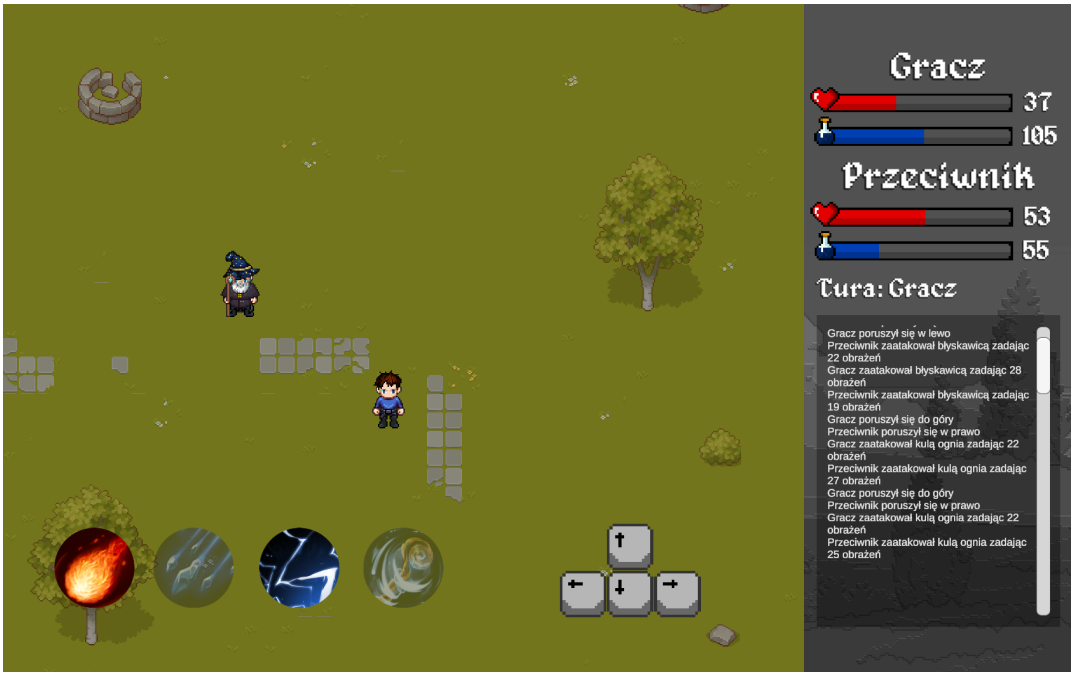
Rysunek 4.2: Ekran rozgrywki po wykonaniu kilku pierwszych tur

Na ekranie rozgrywki można zauważyć, że odległość pomiędzy graczem a przeciwnikiem zmalała do 6 kratek. Zmieniły się również suwaki przedstawiające zasoby postaci. Przeciwnik posiada w tym momencie **150** punktów many, gdyż atak Błyskawica kosztował go tych punktów 50. Gracz posiada **83** punkty życia, gdyż został zaatakowany w poprzedniej turze atakiem Błyskawica, zadającym mu 22 obrażeń, co sprawiło, że jego punkty życia zmalały do wartości **78**. Jednak wraz z początkiem tury gracza zregenerował on swoje statystyki w postaci dodatkowych 5 punktów życia oraz 15 punktów many, dając całościowy wynik punktów życia na poziomie **83**. Na ekranie widać również zapis z konsoli.

Odległość 6 kratek od przeciwnika sprawia, że użytkownik może wykonać swój pierwszy atak – atak Błyskawica. Wysoka liczba punktów życia gracza oraz punktów many przeciwnika sprawia, że w przypadku sztucznej inteligencji przeciwnika wciąż wysoko wartościowanymi atakami będą ataki Kula ognia oraz Błyskawica. Na Listingu 4.2 znajduje się zapis kolejnych 5 tur, a na Rysunku 4.3 widać ekran rozgrywki po kolejnych turach.

```
1 Gracz zaatakował błyskawicą zadając 28 obrażeń
2 Przeciwnik zaatakował błyskawicą zadając 19 obrażeń
3 Gracz poruszył się do góry
4 Przeciwnik poruszył się w prawo
5 Gracz zaatakował kulą ognia zadając 22 obrażeń
6 Przeciwnik zaatakował kulą ognia zadając 27 obrażeń
7 Gracz poruszył się do góry
8 Przeciwnik poruszył się w prawo
9 Gracz zaatakował kulą ognia zadając 22 obrażeń
10 Przeciwnik zaatakował kulą ognia zadając 25 obrażeń
```

Listing 4.2: Zapis konsoli po wykonaniu kolejnych tur rozgrywki



Rysunek 4.3: Ekran rozgrywki po wykonaniu kolejnych tur rozgrywki

Po kolejnych 5 turach rozgrywki można zauważyć, że nastąpiła pierwsza wymiana ciosów pomiędzy graczem a przeciwnikiem. W pierwszej turze gracz zdecydował się zaatakować atakiem Błyskawica, na co przeciwnik mu odpowiedział tym samym atakiem, losując nieznacznie mniejsze obrażenia. W następnej turze gracz zdecydował się zmniejszyć dystans do przeciwnika ruchem do góry. Po zmianie tury na turę przeciwnika gracz posiadał **69** punktów życia, a przeciwnik posiadał **130** punktów many. Ewaluacja ataków przeciwnika wyglądała następująco: **66** punktów dla ataku Błyskawica, **68** punktów dla ataku Kula ognia oraz **51** punktów dla ataku Pocisk lodu. Z powodu zmniejszenia się punktów many przeciwnika oraz punktów many gracza, większą wartość zyskał atak Pocisk lodu, a na wartości straciła pozostała dwójka ataków. Odległość przeciwnika od gracza wciąż wynosiła 6 kratek w poziomie, więc jedynym atakiem, który przeciwnik mógł wykonać był atak Błyskawica. W wyniku losowania przeciwnik wybrał któryś z pozostałych dwóch

ataków. Niemożność wykonania go z powodu zbyt dużej odległości spowodowała ruch przeciwnika w stronę gracza, aby ten dystans zminimalizować.

Jako że odległość gracza od przeciwnika wynosiła już tylko 5 kratek w pionie oraz w poziomie, w turze gracza został odblokowany atak Kula ognia. W tej turze gracz oraz przeciwnik wymienili się tymi atakami z nieznaczną korzyścią dla przeciwnika. W kolejnej turze nastąpiło skrócenie dystansu ze strony gracza oraz przeciwnika, a w ostatniej gracz wraz z przeciwnikiem znów wymienili się atakami Kula ognia.

Rozgrywka trwała jeszcze kilkanaście tur. Postacie wymieniały się atakami oraz zmniejszały dystans wobec siebie. Gdy skończyły się punkty many, gracz oraz przeciwnik skrócili dystans do siebie do jednej kratki, aby móc wykonać atak Uderzenie kijem. Po wymianie ciosów gracz oraz przeciwnik ponownie zwiększyli odległość między sobą i dalej kontynuowali walkę. Gdy punkty życia przeciwnika spadły do niskiej wartości, gracz postanowił oszczędzić swoje punkty many, aby móc zaatakować atakiem Kula ognia. Niestety dwa razy z rzędu wylosował najniższą wartość, dzięki czemu przeciwnik przetrwał atak. Akcją kończącą rozgrywkę był atak przeciwnika atakiem Błyskawica, zadając **21** obrażeń, co w konsekwencji zmniejszyło punkty życia gracza do zera i zakończyło grę. Reszta przebiegu rozgrywki została przedstawiona na Listingu 4.3 na samym końcu rozdziału.

Rysunek 4.4: Ekran końcowy po przegranej gracza

Na Rysunku 4.4 widać ekran końcowy, który został wyświetlony po rozgrywce. Gracz ostatecznie przegrał potyczkę ze sztuczną inteligencją przeciwnika. Na ekranie końcowym można zauważyć napis „Porażka" oraz zasoby obydwu postaci w chwili śmierci gracza. Dodatkowo wyświetlone zostały dwa przyciski. Pierwszy, „Zagraj jeszcze raz" resetuję rozgrywkę i przenosi gracza do menu gry, skąd może rozegrać partię z przeciwnikiem po raz kolejny. Przycisk „Wyjdź" wychodzi z aplikacji.

```
 1 Gracz zaatakował błyskawicą zadając 23 obrażeń
 2 Przeciwnik zaatakował kulą ognia zadając 23 obrażeń
 3 Gracz zaatakował błyskawicą zadając 13 obrażeń
 4 Przeciwnik poruszył się w prawo
 5 Gracz poruszył się do góry
 6 Przeciwnik poruszył się w prawo
 7 Gracz zaatakował błyskawicą zadając 15 obrażeń
 8 Przeciwnik zaatakował błyskawicą zadając 19 obrażeń
 9 Gracz poruszył się do góry
10 Przeciwnik poruszył się w prawo
11 Gracz zaatakował magicznym kijem zadając 9 obrażeń
12 Przeciwnik zaatakował magicznym kijem zadając 8
     obrażeń
13 Gracz zaatakował magicznym kijem zadając 6 obrażeń
14 Przeciwnik poruszył się w prawo
15 Gracz zaatakował kulą ognia zadając 20 obrażeń
16 Przeciwnik zaatakował błyskawicą zadając 13 obrażeń
17 Gracz poruszył się do dołu
18 Przeciwnik poruszył się do góry
19 Gracz poruszył się w prawo
20 Przeciwnik zaatakował pociskiem lodu zadając 15
     obrażeń
21 Gracz poruszył się w prawo
22 Przeciwnik poruszył się do dołu
23 Gracz zaatakował kulą ognia zadając 20 obrażeń
24 Przeciwnik poruszył się do dołu
25 Gracz poruszył się w lewo
26 Przeciwnik zaatakował błyskawicą zadając 21 obrażeń
```

Listing 4.3: Końcowy przebieg rozgrywki

Podsumowanie

Celem niniejszej pracy licencjackiej było pokazanie, jakie możliwości leżą u podstaw symulowania sztucznej inteligencji w grach. Na prostym przykładzie logiki rozmytej zostało pokazane, w jaki sposób można zaprojektować, a następnie zaimplementować zachowanie postaci w grze komputerowej.

Realizacja projektu na pewno wymagała zapoznania się z grami komputerowymi oraz zauważenia, w jaki sposób sztuczna inteligencja wchodzi w interakcję z graczem. Historia sztucznej inteligencji oraz gier, przedstawiona w rozdziale 1, niewątpliwie miała duży wpływ na określenie możliwości realizacji projektu i nadała kierunek całej pracy.

Symulowanie zachowania czy walki postaci stosując logikę rozmytą to tylko jedna z wielu możliwych opcji. Innymi sposobami wartymi rozważenia na pewno są maszyny stanów, drzewa behawioralne czy bardziej zaawansowane algorytmy zachowania. Wszystkie wyżej wymienione opcje są z powodzeniem wykorzystywane w przemyśle gier wideo. Sama logika rozmyta również może zostać wykorzystana na przykład do symulacji fizyki bądź otoczenia gry, a nie tylko do sterowania zachowaniem postaci.

Użycie sztucznej inteligencji zdecydowanie ubogaciło rozgrywkę. Można z całą pewnością stwierdzić, że zaimplementowany w ten sposób model zachowania przeciwnika podwyższył poziom trudności i stanowi o wiele większe wyzwanie, niż gdyby przeciwnik zachowywał się w całości losowo. Kilka prostych reguł połączonych algorytmem nadało duszę postaci i sprawiło, że zachowuje się ona o wiele bardziej racjonalnie.

Stworzona aplikacja niewątpliwie stanowi bazę do dalszej pracy i rozwoju. Można tu chociażby wspomnieć o dodaniu dodatkowych przeciwników, symulowanych innymi mechanikami. Rozbudowa mapy, dodanie poziomów doświadczenia czy zwiększenie liczby możliwych ataków to tylko niektóre z wielu pomysłów, jakie mogą zostać zrealizowane w przyszłości.

Bibliografia

[1] Anders Tychsen, *Role playing games: comparative analysis across two media platforms*, Macquarie University, Department of Computing, 2006.

[2] Matt Barton, *The History of Computer Role-Playing Games Part 1: The Early Years (1980-1983)*, Gamasutra, 2007.

[3] Matt Barton, *The History of Computer Role-Playing Games Part 2: The Golden Age (1985-1993)*, Gamasutra, 2007.

[4] Matt Barton, *The History of Computer Role-Playing Games Part III: The Platinum and Modern Ages (1994-2004)*, Gamasutra, 2007.

[5] Jan Argasiński, *Sztuczna inteligencja w grach wideo*, Wydawnictwo Uniwersytetu Jagiellońskiego, 2012.

[6] Cainos, *Pixel Art Top Down - Basic*, https://cainos.itch.io/pixel-art-top-down-basic (dostęp: 14.09.2023)

[7] *Universal LPC Spritesheet Character Generator*, https://sanderfrenken.github.io/Universal-LPC-Spritesheet-Character-Generator (dostęp: 14.09.2023)

[8] craftpix.net, *Free Nature Backgrounds Pixel Art*, https://craftpix.net/freebies/free-nature-backgrounds-pixel-art/ (dostęp: 14.09.2023)

[9] Hewett Tsoi, *Alagard Font*, https://www.dafont.com/alagard.font (dostęp: 14.09.2023)

[10] PONETI, *Basic RPG Icons*, https://assetstore.unity.com/packages/2d/gui/icons/basic-rpg-icons-181301 (dostęp: 14.09.2023)

[11] Blink, *FREE - RPG Fantasy Spell Icons*, https://assetstore.unity.com/packages/2d/gui/icons/free-rpg-fantasy-spell-icons-200511 (dostęp: 14.09.2023)

[12] ji-hyeeeee, *Awesome Fantasy Skill Icon pack*, https://assetstore.unity.com/packages/2d/gui/icons/awesome-fantasy-skill-icon-pack-147529 (dostęp: 14.09.2023)

[13] Free Game Assets, *Free 50 Aeromancer Skills*, https://free-game-assets.itch.io/free-50-rpg-aeromancer-skill-icons (dostęp: 14.09.2023)

[14] BlatFan, *BlatKeys*, https://blatfan.itch.io/blatkeys (dostęp: 14.09.2023)

[15] Lornn, *Fantasy Forest Backgrounds*, https://lornn.itch.io/fantasy-forest-backgrounds (dostęp: 14.09.2023)

[16] Zed Hanok, *Misc HUD and UI Graphics*, https://zed-hanok.itch.io/misc-hud-and-ui-graphics (dostęp: 14.09.2023)

Spis rysunków

Spis listingów

Spis tabel