
**POLITECHNIKA ŚLĄSKA**

**WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI**

**KIERUNEK: INFORMATYKA**

Praca dyplomowa inżynierska

# Stworzenie zaawansowanego systemu walki w grze cRPG w oparciu o podobieństwo rysowanych przez gracza symboli z wykorzystaniem sieci neuronowych

autor: Igor Budzyński

kierujący pracą: dr inż. Damian Pęszor

Gliwice, styczeń 2022

# Spis treści

# Streszczenie

Uczenie maszynowe jest jedną z gałęzi sztucznej inteligencji, która polega na wykorzystaniu algorytmów wyszukujących korelacje i wzorce w zbiorach danych i na ich podstawie dokonywać najwłaściwszych wyborów. Celem pracy jest stworzenie zaawansowanego systemu walki do gry akcji opartego na rozpoznawaniu narysowanych odręcznie symboli z wykorzystaniem tej technologii. Głównym założeniem pracy, jest sprawienie, by implementowane rozwiązanie było łatwe do wdrążenia w dowolnym projekcie. Praca została wykonana w silniku Unity i posiada testową aplikację, ale może zostać dodana do dowolnego rozwiązania za pomocą paczki. Model sieci neuronowej, która zajmuje się predykcją na narysowanych symbolach, został stworzony przy użyciu ML.NET Model Builder. W trakcie przeprowadzania analiz porównano istniejące rozwiązania oraz publikacje naukowe i na ich podstawie wybrano najlepiej pasujące solucje do tego projektu. Ten system wyróżnia się od reszty swoim nietypowym połączeniem dynamicznej walki z rysowaniem symboli. W projekcie skupiono się na tym by rzucanie zaklęć, co jest nieodłączną częścią tworzonego systemu, dawało graczowi dużo możliwości tworzenia reakcji łańcuchowych między czarami.

**Słowa kluczowe: System walki, cRPG, Uczenie maszynowe, Rysowanie symboli**

# Rozdział 1

# Wstęp

## 1.1 Wprowadzenie do uczenia maszynowego

Uczenie maszynowe jest jedną z gałęzi sztucznej inteligencji, która polega na wykorzystaniu algorytmów i danych tak, aby w naśladujący ludzki sposób nauki zwiększać dokładność uzyskiwanych wyników. Algorytmy wykorzystywane w uczeniu maszynowym są stworzone, tak by wyszukiwać korelacje i wzorce w zbiorach danych i na ich podstawie podejmować najlepsze decyzje. Zaprogramowany model wymaga dużego zestawu danych, na których będzie się uczyć. W nadzorowanym uczeniu maszynowym, które użyte jest w projekcie, każda z próbek danych, na której przygotowuje się model, jest podawana z informacją o poprawnym wyniku, dzięki czemu sieć wie jaki wynik powinna otrzymać. Po wytrenowaniu można użyć kolejnego zestawu danych, aby sprawdzić niezawodność przewidywań sieci neuronowej. Ta technologia coraz częściej znajduje zastosowanie we wszystkich aspektach naszego życia takich jak dobór reklam wyświetlanych na stronach internetowych, prognozy pogody czy nowoczesne roboty domowe.

## 1.2 Charakterystyka walki w grach cRPG

Gry cRPG (computer Role Playing Game) inaczej nazywane fabularnymi to gatunek gier komputerowych, w którym gracz wciela się w bohatera, często stworzonego przez siebie, i przeżywa przygodę w przedstawionym świecie fikcyjnym.

Spośród wielu aspektów rozgrywki zwykle bardzo ważnym elementem jest walka. Na przestrzeni lat, w których gatunek się rozwijał i przyjmował różne formy, można wyróżnić kilka często powtarzających się wzorców. Pierwsze mechaniki walki w grach typu cRPG były podobne bardzo do swojego pierwowzoru, jakim były klasyczne papierowe RPG. Postać miała swoje cechy i to czy udało się wykonać pewną czynność było rozpatrywane na podstawie sprawdzenia czy wartość danej cechy jest wystarczająco wysoka. Poźniej do gier cRPG zawitała losowość znana z swojego pierwowzoru jakim były gry fabularne i sukces zaczął być rozstrzygany przy użyciu liczb pseudolosowych a prawdopodobieństwo sukcesu było zależne od statystyk postaci. Przykładowo statystyka zręczności wpływała na prawdopodobieństwo trafienia bronią strzelecką. Później ten system ewoluował i przyjmował różne formy. Szczególnie w podgatunku acRPG (action computer Role Playing Game) elementy losowe zaczęły być bardziej dodatkiem do samych systemów walki. To czy udało się trafić przeciwnika, było rozstrzygane na podstawie tego jak sprawnie celujemy myszką, a elementem zmiennym były tylko obrażenia, które i tak były zacieśnione górną i dolną granicą. W nowoczesnych grach acRPG walka coraz częściej polega na odpowiednim wyczuciu czasu i refleksie gracza, a rzadziej na losowaniu wyników zdarzeń. Produkcje, których rozgrywka skupia się na walce przy użyciu broni białej posiadają rozróżnienie kierunków ataków i bloków co sprawia, że gracz jeszcze bardziej musi wykazać się zręcznością. Do starszych przedstawicieli gatunku można zaliczyć chociażby „Baldur's Gate" a do nowszych „Wiedźmina 3" czy „Sekiro".

## 1.3 Cel pracy

Celem pracy inżynierskiej jest stworzenie zręcznościowego systemu walki do gry cRPG z wykorzystaniem uczenia maszynowego, a dokładnie sieci neuronowych. Najważniejszą częścią projektu będzie stworzenie algorytmu sztucznej inteligencji, która będzie odczytywać narysowany przez gracza symbol i na jego podstawie rozpoznawać, do którego z typów ataku jest on najbardziej podobny. W grze to wynik rozpoznawania sieci neuronowej będzie decydował, z jaką skutecznością i jakie zaklęcie rzuci pojedynkujący się czarodziej. Ta aplikacja charakteryzuje się systemem walki polegającym na wykorzystywaniu refleksów gracza w połączeniu zręcznością w odwzorowywaniu symboli. Bardzo ważnym elementem jest sprawie-

nie, aby gracz nie rzucał zaklęć bez zastanowienia, w losowej kolejności, dlatego elementy zaklęć będą wchodzić z sobą w reakcje, wywołując mocniejsze lub zmienione efekty. Celem takiego podejścia jest wymuszenie na graczu zastanowienie się jakiej kombinacji najlepiej użyć.

## 1.4 Zakres pracy

**W zakresie pracy inżynierskiej mieści się:**

- Zaprojektowanie działającego systemu walki do gry acRPG.
- Stworzenie modelu sieci neuronowej odpowiedzialnej za interpretacje symboli narysowanych przez gracza.
- Zaimplementowanie sposobu poruszania się postacią gracza opartego na fizyce
- Zaprojektowanie mechaniki wywoływania zaklęć przez gracza.
- Zaprojektowanie różnych technik wykonywania zaklęć
- Zaprojektowanie systemu relacji między czarami, tak by te wchodziły w reakcje łańcuchowe.

**W zakres pracy inżynierskiej nie wchodzą:**

- Stworzenie systemu relacji między graczem i postaciami niezależnymi.
- Stworzenie sztucznej inteligencji dla przeciwników, z którymi gracz będzie się mierzył w pojedynkach.
- Modele i grafiki użyte w grze.
- Mapy oraz tereny, po których będzie poruszać się gracz.
- Programy cieniujące i algorytmy oświetlenia użyte w projekcie.
- Efekty cząsteczkowe zastosowane w aplikacji.

## 1.5 Charakterystyka rozdziałów

Poniżej znajduje się krótka charakterystyka każdego z 7 rozdziałów.

1. **Wstęp -** w tym rozdziale znajduje się wprowadzenie do tematyki pracy. Zaznaczone jest, jakiej dziedziny technologii dotyczy projekt oraz jaki jest jego cel. Zarysowana jest tutaj koncepcja projektu i krótki opis każdego z rozdziałów.

2. **Analiza tematu -** w tym rozdziale znajduje się wprowadzenie do poruszanego w pracy problemu, rozważania teoretyczne na jego temat, a także dostępne już, przeprowadzone analizy i rozwiązania z publikacji naukowych, książek i artykułów popularnonaukowych.

3. **Wymagania i narzędzia -** w tym rozdziale zdefiniowane zostały wymagania funkcjonalne i niefunkcjonalne, opis użytych narzędzi, a także metodyka, jaką podjęto podczas pracy nad implementacją projektu.

4. **Specyfikacja zewnętrzna -** w tym rozdziale wypisane zostały wymagania sprzętowe oraz programowe, których spełnienie jest kluczowe do uruchomienia aplikacji testowej opisywanego systemu walki. Zawarto tu także sposób obsługi systemu, którego działanie zaprezentowane jest na załączonych zdjęciach.

5. **Specyfikacja wewnętrzna -** w tym rozdziale scharakteryzowana została wewnętrzna struktura projektowanego systemu, a także przegląd użytych bibliotek i komponentów. Dodatkowo zawarte są tutaj opisy najważniejszych klas programu, użyte w rozwiązaniu wzorce projektowe oraz diagramy UML.

6. **Weryfikacja i walidacja -** w tym rozdziale przedstawiono, jak aplikacja była testowana i poprawiana w procesie tworzenia. Znajdują się tutaj wypisane znane błędy, jakie znaleziono w czasie implementacji rozwiązania oraz analiza stwierdzająca, czy wszystkie wymagania zostały spełnione.

7. **Podsumowanie i wnioski -** w tym rozdziale znajduje się podsumowanie całej pracy nad systemem, w tym: wyciągnięte wnioski, napotkane błędy oraz możliwości rozwoju rozwiązania w przyszłości.

# Rozdział 2

# Analiza tematu

## 2.1 Wprowadzenie do dziedziny

Systemy walki są w ostatnich latach prawie niezbędnym elementem każdej dużej gry cRPG. Jest tak dlatego, że gry akcji cieszą się teraz ogromną popularnością, co pokazuje ranking TOP gier na PC sporządzony przez platformę Ceneo, w którym 5 na 10 pozycji zajmują gry acRPG[1]. Walka będąc tak ważnym elementem składowym dużej części gier fabularnych, przybierała różne formy w zależności od założeń danej produkcji acRPG. Można zauważyć zależność, że z im większą liczbą przeciwników mierzy się gracz, tym mniej systemy opierają się na zręczności gracza tzn. wyczuciu czasu czy refleksie. To właśnie w grach akcji skupiających się na walce z niedużą grupą przeciwników ważne są odpowiednio wyczekane bloki ataków, czy wymierzone uniki ciosów.

Zgodnie z analizą przeprowadzoną przez Interactive Institute [8] gry cRPG można podzielić na podstawie zaprojektowanego systemu walki. Badania te przeprowadzono na 41 produkcjach, co pozwoliło na podział gier cRPG na liczne podgrupy, z których głównymi są Action-Oriented cRPGs (gry fabularne o tematyce akcji) i Tactical cRPGs (taktyczne gry fabularne). Dokładne graficzne przedstawienie analizy widoczne jest na Rysunku 2.3. Gra cRPG, do której zostanie użyty system walki będący celem tego projektu, będzie należeć do podkategorii CBR cRPGs (Constant Battle Risc cRPGs), lub będzie jej niewymienionym w podziale potomkiem. Jest tak dlatego, że walka w grze będzie się skupiać na pojedynkowa-

niu z pojedynczym przeciwnikiem, który będzie posiadał wiele możliwości wyprowadzaniu ataków. Istotnym elementem tworzonego systemu jest sprawienie, aby mechanika walki była wymagająca, co jest kolejną cechą CBR cRPGs.

Używanie rozwiązań opartych na uczeniu maszynowym w grach komputerowych staje się coraz bardziej popularne, a najczęściej pojawiają się one przy konstruowaniu sztucznej inteligencji przeciwników. W przypadku tego projektu potrzebne jest inne, rzadko spotykane w grach zastosowanie sieci neuronowej, którym jest rozpoznawanie odręcznie narysowanych symboli. To jak precyzyjny będzie model sieci neuronowej, zależy od: wielkości zestawu danych, użytych algorytmów uczenia i funkcji aktywacyjnych, a także ilości warstw ukrytych.

W publikacji [3], gdyż wydanej w 2020 roku, został zaprojektowany model sieci neuronowej przy użyciu pythonowej biblioteki Sklearn. Model ten został kilkakrotnie przetrenowany przy użyciu różnych algorytmów klasyfikujących. Trafność i wyniki F1 algorytmów użytych we wspomnianej analizie widoczne są na rysunkach 2.1 i 2.2. Zgodnie z wynikami w obydwóch raportach najwyższe wyniki osiągnął SVM (Support Vector Machines), który jest jednym z najczęściej używanych algorytmów w problemach związanych z analizą obrazów, a drugim, niewiele odbiegającym od SVM, jest KNN (K Nearest Neighbours).

## 2.2 Założenia projektowanego systemu walki

Założeniem projektowanego systemu jest umożliwienie wzięcie udziału w pojedynku dwóch magów walczących z sobą w czasie rzeczywistym. Gracz ma pełną możliwość ruchu, w którym poza zwyczajnym chodzeniem może wykonywać takie akcje jak sprint, skok czy uniki. W czasie walki gracz może zatrzymać się i za pomocą myszki narysować symbol, który później zostanie zinterpretowany przez sieć neuronową. Jeśli podobieństwo znaku będzie bliskie ideału, to zostanie wywołane zaklęcie przypisane do symbolu, które gracz może użyć przeciwko swojemu wrogowi. Poprawnie rzucone zaklęcia mogą nałożyć na postaci lub otoczenie odpowiednie magiczne efekty. Jeśli odwzorowany przez gracza znak nie jest wystarczająco podobny do tego, co sieć neuronowa uznaje za poprawne, to zostanie na niego nałożony negatywny efekt. Bardzo istotnym elementem projektowanego

| Classifiers | Accuracy Score |
|---|---|
| SVM | 0.9688 |
| KNN | 0.9555 |
| Stochastic Gradient Descent | 0.8932 |
| Naïve Bayes | 0.8075 |
| Random Forest | 0.7532 |
| Decision Trees | 0.7352 |

Rysunek 2.1: Trafność algorytmów klasyfikujących użytych w modelu sieci neuronowej rozpoznającej odręcznie narysowane cyfry.[3]

| Classifiers | F1-Score |
|---|---|
| SVM | 0.92 - 0.99 |
| KNN | 0.90 - 0.99 |
| Stochastic Gradient Descent | 0.81 - 0.97 |
| Naïve Bayes | 0.68 - 0.97 |
| Random Forest | 0.41 - 0.93 |
| Decision Trees | 0.64 - 0.91 |

Rysunek 2.2: F1-Score algorytmów klasyfikujących użytych w modelu sieci neuronowej rozpoznającej odręcznie narysowane cyfry.[3]

systemu jest sprawienie, aby efekty wywołane przez zaklęcia wchodziły w reakcje łańcuchowe z sobą. Przykładowo, podpalony przeciwnik, gdy zostanie trafiony odpowiednim zaklęciem typu wodnego, może zostać ugaszony.

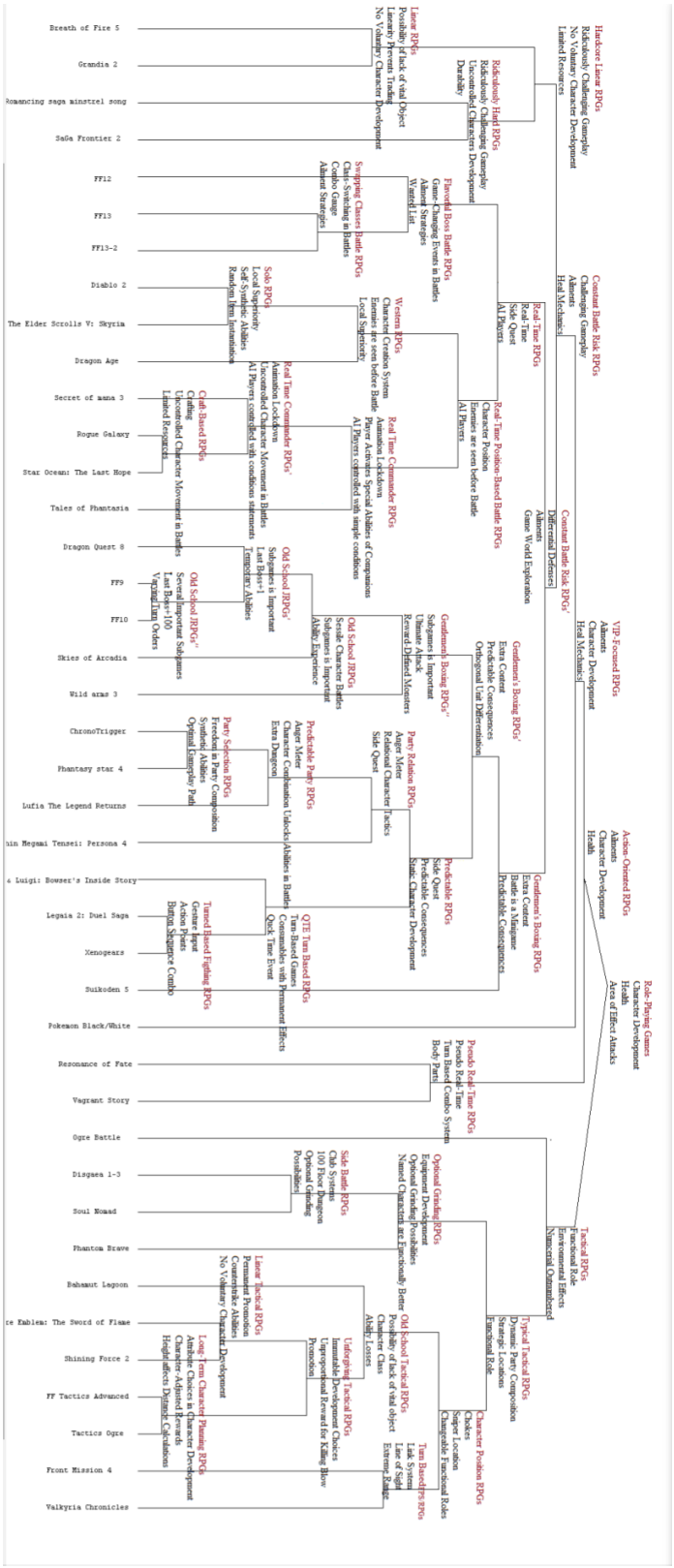
## 2.3 Przegląd literatury

W ostatnich latach coraz bardziej rozwija się rynek gier, co pokazują badania [9]. Graczy jest coraz więcej, powstają nowe przedsiębiorstwa tworzące komputerową rozrywkę, a istniejące się rozrastają. Wpłynęło to na rozwój nowych rozwiązań oraz na ilości prac naukowych i badań traktujących o samych grach komputerowych, istniejących rozwiązaniach znanych problemów czy ich pozytywnym lub negatywnym wpływie na życie ludzi.

Została opublikowana duża ilość artykułów traktująca o systemach walki w grach komputerowych. Począwszy od traktujących o potyczce w klasycznych grach arkadowych na konsole Atari [7], a kończąc na bardziej złożonych grach akcji [5].

### 2.3.1 The Fundamental Pillars of a Combat System

W swojej pracy „The Fundamental Pillars of a Combat System”[5] Sébastien Lambottin opisuje jakie cechy powinien posiadać system walki w grze akcji. Autor stwierdza, że stworzenie dobrej mechaniki nie jest łatwym zadaniem, ale przy odpowiedniej analizie możliwe jest zaprojektowanie rozwiązania, które będzie zapewniać rozrywkę przez wiele godzin. Zgodnie z przytoczonym artykułem zaprojektowanie wielu możliwości ataków sprawia, że rozgrywka jest dużo mniej monotonna, co pozytywnie wpływa na wydłużenie czasu zainteresowania grą. Kolejną rzeczą o jakiej mówi autor, to zadbanie by gracz zawsze posiadał sposób obrony przed atakami przeciwnika, czy to poprzez blokowanie, czy unik wykonany w odpowiednim czasie. Sugerowanym rozwiązaniem jest także sprawienie, by nie wszystkie ataki skupiały się tylko i wyłącznie na zadawaniu bezpośrednich obrażeń, ale by wprowadzały pewne efekty takie jak: ogłuszenia, oślepienia, odepchnięcia, regeneracje zdrowia czy zadawanie obrażeń czasowo. Z doświadczeń opisanych przez Sébastiena Lambottin wynika, że jedną z najbardziej zachęcających rzeczy do dalszej rozgrywki jest pozwolenie, aby gracz czuł się mądrym. Walka w grze, z mechaniką zawierającą




Rysunek 2.3: Podział gier gatunku cRPG pod względem klasyfikacji systemu walki [8]

wiele opcji ataku pozwala na wynalezienie metod na przechytrzanie przeciwnika, co daje dużą satysfakcję z gry.

### 2.3.2 Comparison Of Learning Algorithms For Handwritten Digit Recognition

Zgodnie z wnioskami, do których doszedł Yann LeCun w swojej pracy „Comparison Of Learning Algorithms For Handwritten Digit Recognition" [6] to jaki algorytm klasyfikujący jest najlepszy, jest bardzo względne. Ilość danych, powtórzeń, ukrytych warstw i nawet sam sposób zapisu danych do nauki wpływa na poprawność predykcji przygotowywanego modelu. Autor stwierdza w swojej pracy, że dziedzina ta jest bardzo szybko rozwijającą się gałęzią informatyki, przez co kilka lat przed napisaniem artykułu zupełnie inne algorytmy były uważane za najlepsze niż te, które zostały wytypowane w momencie opublikowania analizy. Przenosząc na szybkość rozwoju dziedzin informatyki, można założyć, że teraz inne algorytmy będą uznawane za najbardziej precyzyjne w przypadku rozpoznawania odręcznie pisanych znaków.

### 2.3.3 Neural Network Model of Artificial Intelligence for Handwritting Recognition

W swojej publikacji „Neural Network Model of Artificial Intelligence for Handwritting Recognition" [4] Shanna Kulik rozważa problemy związane z interpretacją odręcznie narysowanych symboli przez sztuczną inteligencję. Autorka analizuje istniejące rozwiązani, gdzie przy użyciu bazy danych symboli zawierającej 60000 próbek do trenowania i 10000 do testowania, udało się utworzyć model, którego trafność predykcji wynosiła 99.2%. W swojej pracy Shanna Kulik stara się jednak za pomocą sztucznej inteligencji uzyskać dodatkowe informacje z charakteru rysowanych ciągów znaków takie jak: wiek, płeć czy stan psychiczny osoby piszącej.

## 2.4 Analiza istniejących rozwiązań

### 2.4.1 Star Wars: Knights of The Old Republic

System walki w klasycznej grze cRPG „Star Wars: Knights of The Old Republic" był turowy i nie należał do najbardziej skomplikowanych. Całość gry odbywała się w czasie rzeczywistym jednak gdy drużyna gracza została zauważona przez przeciwników, rozgrywka zatrzymywała się i oczom użytkownika ukazywał się panel przedstawiony na Rysunku 2.4. Postacią gracza jest osoba stojąca plecami do kamery, natomiast jego wrogiem istota stojąca w głębi korytarza, nad którą wyświetla się czerwony panel z podpisem "Vulkar Patrol Droid". Gracz może zaplanować w przyszłość swoje akcje, aż do 4 rund. Mogą to być ataki, ich warianty wyświetlone są w lewym okienku czerwonego panelu nad wrogiem, użycie mocy lub granatów, których opcje znajdują się w kolejnych panelach. Poza tym gracz może zaplanować zastosowanie paczek leczniczych lub tarcz. Gdy gracz stwierdzi, że jest gotowy do walki, to może odmrozić czas, a jego postać przejdzie do wykonywania zaplanowanych akcji, w tej samej kolejności, w jak były wybierane. Przeciwnik gracza również rozpocznie swoje wrogie akcje, które będą następować na zmianę z operacjami postaci grywalnej. W ten sposób system walki stworzony w KOTOR (Knights of The Old Republic) w pewnym sensie nie jest zwyczajnym systemem turowym, a jednak pozostawia wiele z jego założeń.

### 2.4.2 Wiedźmin 3: Dziki Gon

W grze fabularnej „Wiedźmin 3: Dziki Gon" walka przeprowadzana jest w czasie rzeczywistym. Ta produkcja jest jedną z najbardziej znanych przedstawicieli gier z bardzo złożonym systemem walki o charakterze zręcznościowym. Podczas rozgrywki poza kilkoma wariantami słabych i silnych ataków mieczem, gracz ma do dyspozycji: kuszę, dwa rodzaje uników, kilka rodzajów magii pomocniczej zwanej znakami, dużą ilość mikstur wywołujących efekty pomocnicze, a także cały arsenał przeróżnych petard i olejów do oręża, których każdy działa na pewien rodzaj przeciwników i może wspomóc w potyczce z nimi. Zwycięstwo w potyczce z wrogiem można uzyskać, wykazując się zręcznością i refleksem. Należy uniknąć zadawanych przez wrogów ciosów, a także używać tej części ekwipunku wiedźmina, na jaką


Rysunek 2.4: Zatrzymanie czasu przed walką w Star Wars: Knights of The Old Republic.

część wykazywał słabość jego przeciwnik. Gracz w trakcie rozgrywki ma możliwość ulepszania wiedźmińskiego sprzętu, a także rozwijanie umiejętności związanych z magią lub posługiwaniem się mieczem. Taka ilość możliwości sprawiła, że wręcz niemożliwym jest dla przeciętnego gracza zbadanie każdego aspektu systemu walki dokładnie, ale tym samym sprawiło, że gra została wyróżniona tytułem Gry Roku, a rozgrywka nie nudziła się graczom.

### 2.4.3 Harry Potter i Kamień Filozoficzny

Rysowanie symboli w grach mobilnych jest dość często spotykanym rozwiązaniem. Zwykle prezentuje się je w grach logicznych, gdzie gracz ma narysować jakiś znak, by przejść do następnej układanki. Znalazło to swoje zastosowanie też w grze na telefony i tablety, której akcja odbywa się w uniwersum Wiedźmina, gdzie podczas walki z potworem gracz może rysować magiczne symbole. Rysowanie symboli bardzo rzadko pojawia się, jednak w grach na komputery osobiste i konsole, z powodu, że rysowanie myszką lub gałką kontrolera jest niewygodne i nieprecy-


Rysunek 2.5: Walka z czterema przeciwnikami w grze Wiedźmin 3: Dziki Gon.

zyjne. Jednym z niewielu przedstawicieli gier komputerowych wykorzystujących to rozwiązanie jest „Harry Potter i Kamień Filozoficzny". W tym tytule niektóre obiekty gry były oznaczone znakami jak na Rysunku2.6, co oznaczało dla gracza, że na tym przedmiocie może wykonać akcję uruchamiającą zaklęcie. Aby uaktywnić czar, użytkownik musiał za pomocą myszki odrysować symbol i jeśli zrobił to odpowiednio dokładnie, to uaktywniał efekt.

## 2.5 Wnioski z przeglądu literatury i istniejących rozwiązań

Systemy walki w grach cRPG potrafią być bardzo różnorodne, można nawet wysnuć stwierdzenie, że każdy z przedstawicieli tego gatunku różni się od pozostałych na samej podstawie tego, jak przeprowadzana jest potyczka z przeciwnikami. Pierwszym rozważanym w tym rozwiązaniu aspektem był wybór czy walka powinna być turowa, czy odbywać się w czasie rzeczywistym. Jednym z założeń projektowanego systemu miało być zachowanie dynamicznych starć i stawianie na refleks i zręczność gracza, więc walka powinna być prowadzona bez przerw w

Rysunek 2.6: Rysowanie magicznego symbolu w grze Harry Potter i Kamień Filozoficzny.

czasie rzeczywistym. Kolejną rzeczą przemawiającą za tym wyborem jest fakt, że rozwiązanie ma łączyć mechanikę zręcznościową charakteryzującą gry CBR cRPG (Constant Battle Risc cRPGs) z rzadko spotykanym systemem rysowania za pomocą myszki, który z założenia będzie wymagał dokładności i precyzji co może znacząco spowolnić rozgrywkę.

Rysowanie również może zostać zaimplementowane na kilka sposobów.Inspirując się rozwiązaniem z Harry'ego Pottera, można zaprojektować system, w którym narysowany symbol będzie zapisywany jako obraz i uruchamiał na rysunku predykcję wytrenowanego modelu sieci neuronowej i na tej podstawie zwracał informację z klasyfikacji, jaki znak został naszkicowany i z jaką dokładnością. W przypadku rozwiązania z Harry'ego Pottera najpierw musiałby zostać wybrany symbol do odrysowania, na przykład przy użyciu klawiszy od 0 do 9, a dopiero potem gracz mógłby się skupić na odrysowywaniu danego znaku. Podejście z siecią neuronową nie ma tego problemu, gdyż decyzja o tym jaki znak został narysowany, będzie podjęta podczas predykcji modelu, a nie przez użytkownika. Dodaje to nie tylko element niewiadomej, jaki czar zostanie rzucony, ale też możliwość zaprojektowania większej ilości zaklęć, niż by to było możliwe, gdyby musiały być wywoływane przez odpowiednie klawisze na klawiaturze.

# Rozdział 3

# Wymagania i narzędzia

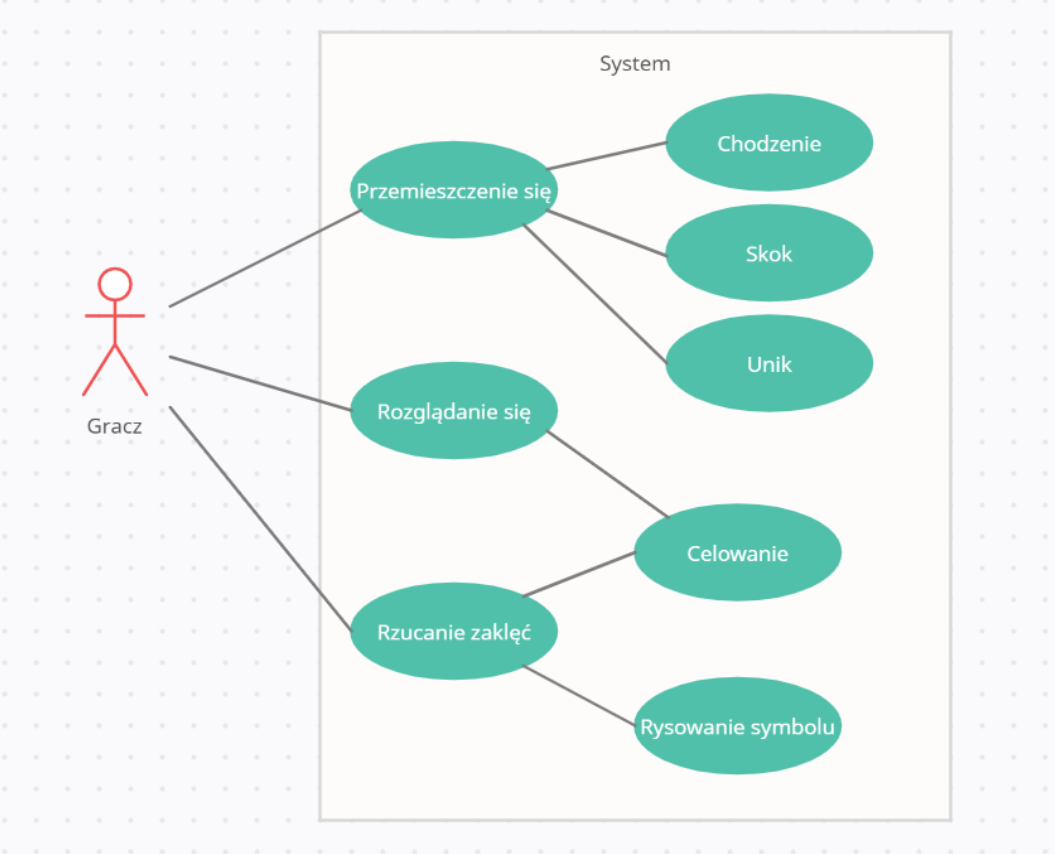
## 3.1 Wymagania

W poniższych podsekcjach przedstawione zostały wymagania funkcjonalne i niefunckjonalne projektowanego systemu.

### 3.1.1 Wymagania funkcjonalne

Tworzony system walki ma następujące wymagania funkcjonalne.

- Gracz ma możliwość poruszania się w wybranych przez niego kierunkach.
- Gracz może wykonać skok.
- Gracz może wykonać unik.
- Gracz może się rozglądać i jednocześnie celować za pomocą myszki.
- Gracz może narysować symbol za pomocą myszki.
- Po narysowaniu symbolu, gracz może uaktywnić jedno z 10 zaklęć, z których każde przypisane jest do jednego znaku.
- Do systemu załączona jest instrukcja odnośnie sterowania.




Rysunek 3.1: Diagram przypadków użycia opisywanego systemu walki.

### 3.1.2 Wymagania niefunkcjonalne

Tworzony system walki ma następujące wymagania niefunkcjonalne.

- System przewidziany jest do stosowania w projektach gier komputerowych.
- System jest zaprojektowany do używania w walkach z niedużą liczbą przeciwników.
- Sieć neuronowa rozpoznająca narysowane symbole jest uruchamiana w osobnej aplikacji dołączonej do projektu.
- Czas rzucania, zapisy obrazu do klasyfikacji i odczytu predykcji w systemie trwa nie więcej niż 0,75 sekundy.

## 3.2 Opis narzędzi

### 3.2.1 Unity

Silnik gier Unity umożliwia tworzenie aplikacji w 2D i 3D na różne platformy. Używa on edytora, który ułatwia projektowanie gier dzięki interaktywnemu podglądowi. Unity posiada bogaty zestaw konfiguracji projektu, który ułatwia przenoszenie aplikacji na inne platformy. Silnik ten jest jednym z najczęściej używanych przez programistów gier przede wszystkim dlatego, że jest darmowy i dość łatwy w nauce, daje możliwość projektowania zarówno aplikacji 2D, jak i 3D, oraz posiada bogatą bazę gotowych zasobów (ang. assets), czyli użytych w projekcie materiałów o różnym charakterze. Mogą to być między innymi tekstury, modele czy nawet gotowe skrypty. Te gotowe zasoby są udostępniane przez innych programistów Unity w sklepie zwanym „Asset Store" i mogą być darmowe lub płatne. Korzystanie z nich jest dużym ułatwieniem dla użytkowników silnika, gdyż rozwiązania przygotowane przez innych mogą oszczędzić nieraz miesiące pracy nad projektem. Program w Unity pisany jest w języku C# w formie skryptów, czyli zwykle pojedynczych klas, które są przetwarzane przez silnik. Unity zostało wybrane jako silnik, w którym powstanie opisywany system walki głównie z powodu bardzo obszernych wbudowanych funkcjonalności niedostępnych domyślnie w innych środowiskach takich jak np. rysowanie na ekranie.

### 3.2.2 Visual Studio 2019

Visual Studio to zintegrowane środowisko programistyczne, którego twórcą jest firma Microsoft. Służy do tworzenia programów na różne platformy w wielu językach, z których pochodzące z rodziny C są domyślnie wspierane. Środowisko to zawiera wiele przydatnych funkcjonalności między innymi: sprawdzanie poprawności kodu w czasie jego pisania, podpowiadanie składni w zgodzie z przyjętą konwencją, a także dowolność w układzie i personalizowaniu okien dialogowych. Visual Studio pomaga utrzymać czystość i przejrzystość programu poprzez podkreślanie nieużytych, a zadeklarowanych zmiennych. Wielką zaletą tego środowiska jest to, że zawiera dużą ilość dodatkowych zawartości i rozszerzeń, które mogą skrócić lub ułatwić pracę programiście. Jednym z najistotniejszym dla tego projektu do-

datkiem jest umożliwiający skompilowanie i używanie powyższych funkcjonalności podczas pisania skryptów w silniku Unity. Co więcej, możliwe jest także debugowanie wybranych skryptów poprzez zatrzymywanie i wznawianie gry w wybranym momencie, a także podgląd zmiennych w zatrzymanych klatkach.

### 3.2.3 Git

Git to system służący do kontroli wersji, który rejestruje zmiany w plikach. Dzięki temu narzędziu możliwe jest przywrócenie historycznej wersji pisanego programu w przypadku wystąpienia błędów lub zmiany sposobu implementacji danego rozwiązania. Git jest bardzo użytecznym narzędziem podczas pracy w zespole, gdyż umożliwia łatwe łączenie napisanego przez kilku programistów programu. Zmiany wprowadzone w projekcie przechowywane są w repozytorium, który jest podzielony na gałęzie — wskaźniki na zestawy zmian. Licencja GNU, na której Git został opublikowany, sprawia, że system jest darmowy do użytku prywatnego i komercyjnego, a jego wielką zaletą jest to, że posiada wsparcie wśród wielu programów takich jak Visual Studio. Posiada on czytelny graficzny interfejs użytkownika (GUI), który sprawia, że korzystanie z niego jest sprawne i proste.

### 3.2.4 ML.NET Model Builder

ML.NET Model Builder to graficzne rozszerzenie do Visual Studio służące do zaprojektowania, wytrenowania i wdrażania modeli uczenia maszynowego. Tworzone scenariusze mogą interpretacje danych rynkowych czy choćby rozpoznawanie elementów obrazów. Model Builder stosuje AutoML (automatyczne uczenie maszynowe), co umożliwia stosowanie różnych algorytmów i ustawień, dzięki którym można drogą testowania znaleźć takie, które najbardziej odpowiadają wybranemu scenariuszowi.

## 3.3 Metodyka pracy nad projektem i implementacja

Pierwszym krokiem pracy nad projektem był przegląd publikacji naukowych i artykułów popularnonaukowych. Następnie sprecyzowane zostały wymagania funkcjonalne i niefunkcjonalne, a także dobrane narzędzia, w których zostanie zaprojektowany system. Kolejnym krokiem było stworzenie wstępnego diagramu klas i przygotowanie środowiska programistycznego. Następnie praca nad projektem została rozłożona na etapy, w jakiś będzie realizowana. W kolejnym etapie przystąpiono do realizacji wcześniej rozplanowanych zadań zgodnie z przyjętym diagramem klas. Pierwszym etapem implementacji kodu było zaprojektowanie i wytrenowanie dzięki rozszerzeniu Model Builder odpowiedniego modelu sieci neuronowej. Drugim krokiem było stworzenie aplikacji, która będzie wywoływana z gry by spełnić następujące akcje: odczyta zapisany obraz, wczyta model sieci, wykona na nim predykcję i na koniec zapisze uzyskane wyniki w pliku binarnym. Kolejnym krokiem było stworzenie projektu w silniku gier Unity oraz umożliwienie graczowi przemieszczania się i rysowania obrazów, które po zapisie miały być przekazywane poprzedniej aplikacji, aby wykonać na nich predykcję. Końcowym krokiem było zaprojektowanie zaklęć i efektów, które mają się wywołać po odczytaniu przewidywań modelu. Wszystkie zmiany w projekcie były monitorowane za pomocą systemu kontroli wersji Git.

# Rozdział 4

# Specyfikacja zewnętrzna

## 4.1 Wymagania sprzętowe i programowe

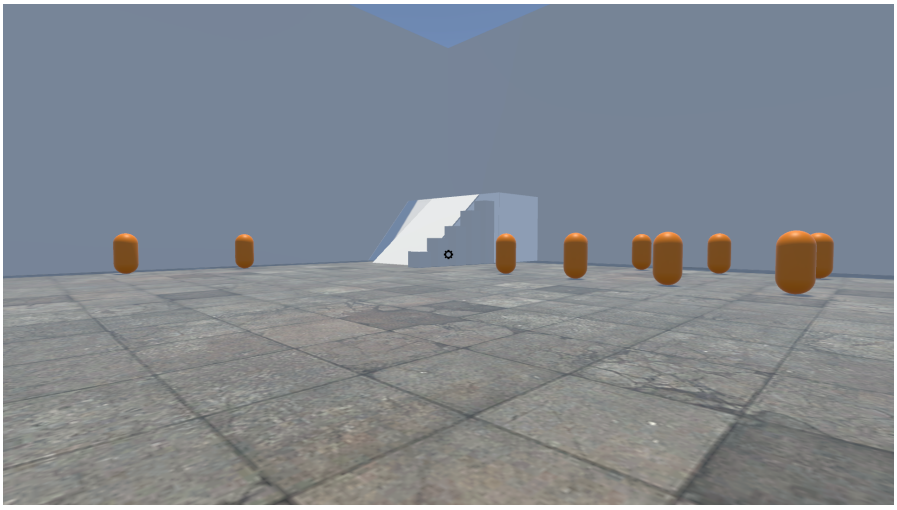
Opisywany w pracy system walki został zaprezentowany w formie małej aplikacji komputerowej, na podstawie której może zostać zbudowana pełna gra cRPG. Aby uruchomić aplikację, niezbędne jest posiadanie minimum 60 MB wolnego miejsca w pamięci komputera.

## 4.2 Instalacja i aktywacja

Aby zainstalować próbną aplikacje demonstrującą zaprojektowany system walki, wystarczy rozpakować całą zawartość paczki o nazwie „WieżaMagów.zip", a następnie uruchomić ją za pomocą pliku „WiezaMagow.exe" znajdującego się w głównym folderze rozpakowanej paczki.

## 4.3 Sposób obsługi

Próbna aplikacja nie posiada żadnego menu i po uruchomieniu gracz od razu pojawia się w pokoju pokazowym zaprezentowanym na zdjęciu 4.1. Obszar ten posiada elementy, które pozwolą użytkownikowi w prosty sposób przetestować wszystkie elementy zaimplementowane w systemie walki. W jednym z narożników pokoju znajdują się platformy umożliwiające przetestowanie mechaniki spadów, krawędzi,



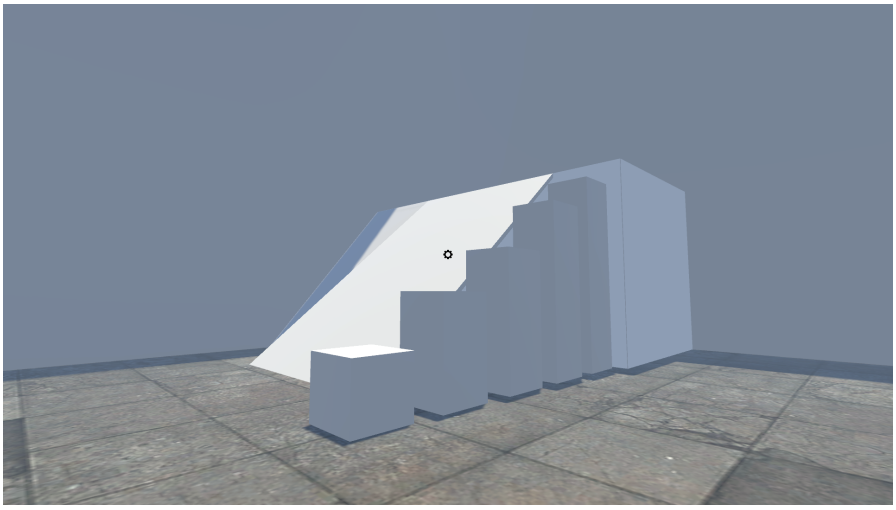
Rysunek 4.1: Widok na całą salę testową aplikacji próbnej

a także skakania. Cały narożnik jest widoczny na ilustracji 4.2. Obok nich w dwóch rzędach ustawieni są przeciwnicy zaprezentowani na zdjęciu 4.3. Jedni to rzucający zaklęcia, inni to po prostu stojące kukły. Ci wrogowie ustawieni są tak, by gracz mógł zaobserwować działanie każdego zaklęcia z perspektywy obserwatora. Po drugiej stronie platformy ustawieni są dwaj wrogowie, którzy są celami dla gracza, aby mógł na nich zobaczyć działanie swoich zaklęć, a także odczuć efekty czarów na sobie.

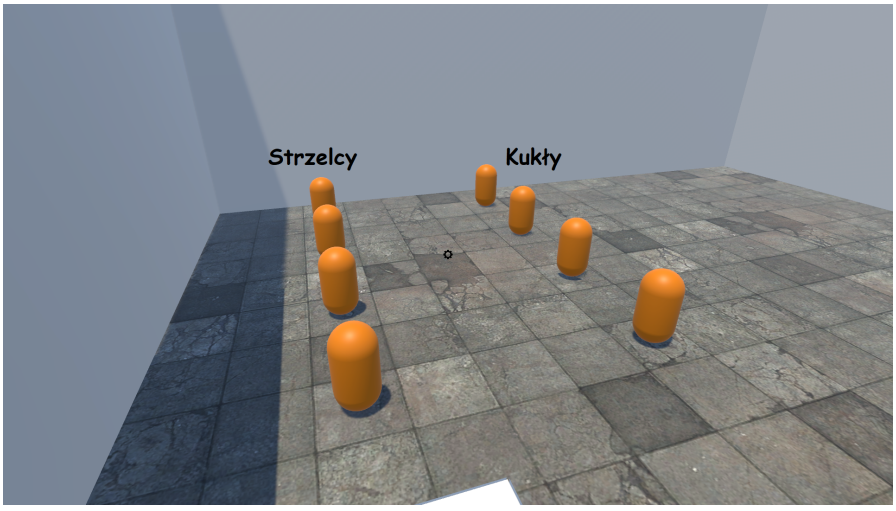
Gracz może poruszać się przy użyciu klawiszy WASD, a także rozglądać się za pomocą myszki. Te podstawowe operacje pozwalają graczowi przemieszczać się w czasie walki, wybierać dogodne pozycje i przy okazji eksplorować całe pomieszczenie testowe.

Aby usprawnić poruszanie się po zróżnicowanym terenie areny, na której odbywać będą się walki, gracz posiada też możliwość skakania. Dzięki temu potrafi pokonywać przeszkody takie jak małe zasłony, czy też ustawiać się w dogodnych pozycjach. Dodatkowo skoki dają graczowi możliwość przeskakiwania nad zaklęciami przeciwników, aby uniknąć ich niechcianego efektu. Aby wykonać skok należy nacisnąć przycisk Spacja na klawiaturze.

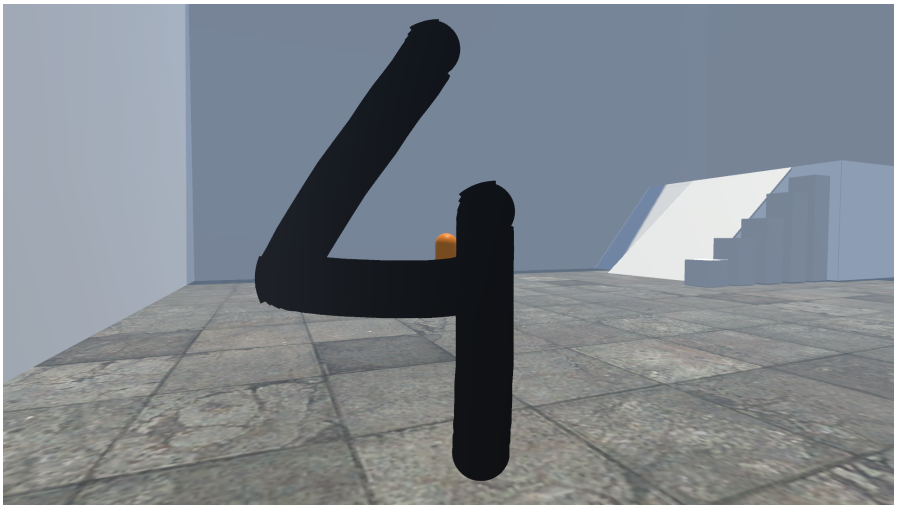
Kolejną zaimplementowaną funkcją jest unik. Daje on graczowi możliwość uskakiwania przed pociskami wystrzeliwanymi przez przeciwników. Po wciśnięciu klawi-



Rysunek 4.2: Platforma umieszczona w rogu sali testowej




Rysunek 4.3: Rząd przeciwników rzucających zaklęcia stojący na przeciwko kukieł

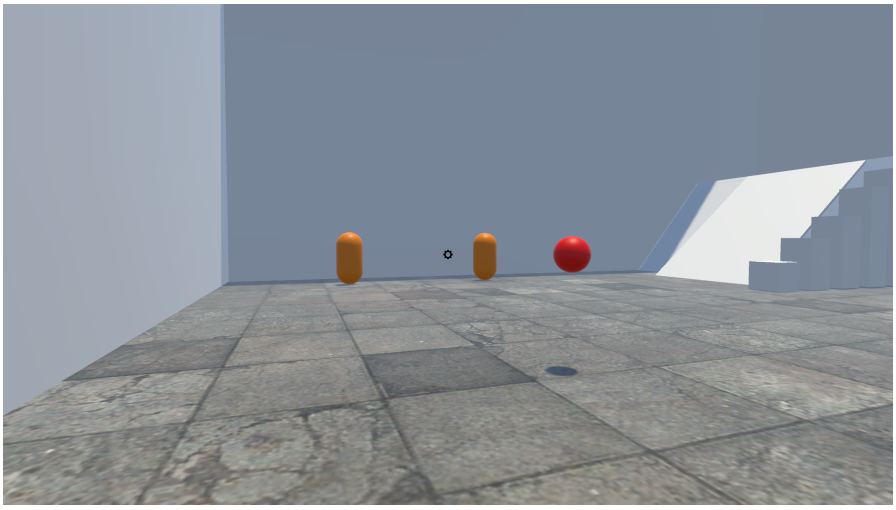


Rysunek 4.4: Przykładowy rysowany symbol w systemie walki

sza Ctrl i pokazaniu dowolnego kierunku za pomocą klawiszy ruchu postać gracza przyspieszy, wykonując poziomy ruch, tak jakby odskakiwała. Ta funkcjonalność pozwala też graczowi rozpędzić się zaraz przed skokiem, dzięki czemu postać jest w stanie przebyć większy obszar w powietrzu. Unik jest ograniczony czasem wyciszenia, który uniemożliwia używanie tej umiejętności cały czas.

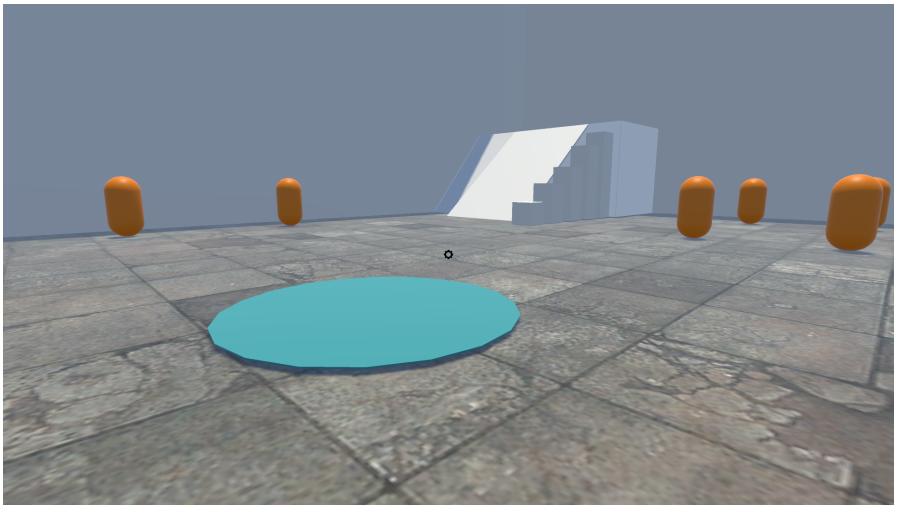
Aby umożliwić graczowi szybkie przebycie otwartych przestrzeni na arenie została dodana opcja sprintu, który uaktywnia się, gdy klawisz Shift jest wciśnięty. Sprint w systemie przejawia się przez zwiększenie szybkości ruchu postaci.

Najistotniejszą funkcją w opisywanym systemie walki jest rysowanie symboli. Gracz jest w stanie w dowolnym czasie rozpocząć rzucanie zaklęcia, wciskając i przytrzymując klawisz Alt. Wtedy na środku ekranu pojawi się kursor wskazujący aktualną pozycję myszy, którą będzie nakreślany symbol tak jak na Rysunku 4.4. Aby zacząć rysować należy przytrzymać lewy przycisk myszy i poruszać myszką tak jak w najprostszym programie do rysowania. Aby zakończyć proces rysowania, należy puścić klawisz Alt. Wtedy zniknie kursor odpowiedzialny za szkicowanie, a pojawi się zakotwiczony na środku ekranu, inny, odpowiedzialny za celowanie. Podczas rysowania gracz nie może się rozglądać ani się poruszać. Jeśli przeciwnik zdoła wytrącić z miejsca osobę nakreślającą symbol, to czar od razu kończy się porażką.

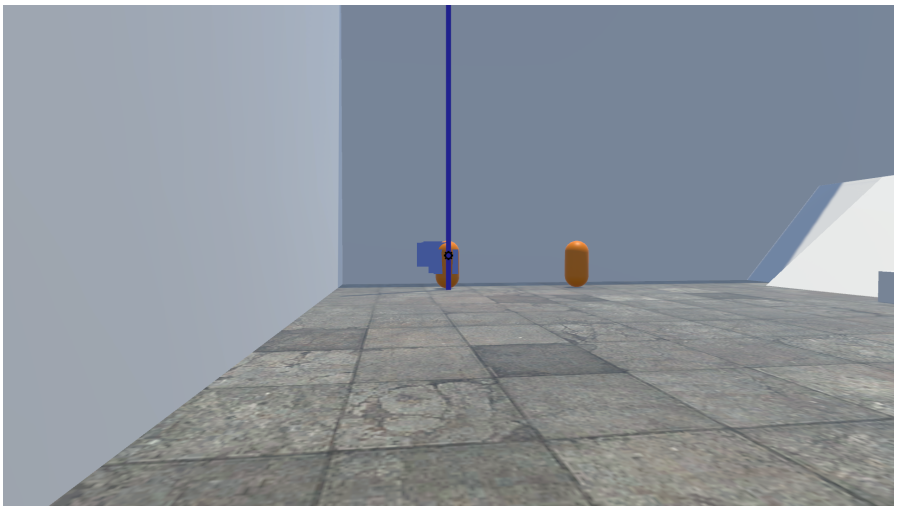


Rysunek 4.5: Zaklęcie „Kula Ognia” rzucone przez gracza

Jeśli symbol został poprawnie odwzorowany, to gracz może uwolnić zaklęcie poprzez kliknięcie lewego przycisku myszy. W przeciwnym wypadku na gracza zostanie nałożony losowy negatywny efekt. Kilka przykładowych zaklęć można zobaczyć na Rysunkach przedstawiających czary Kula Ognia 4.5, Plama Wody 4.6 oraz Piorun 4.7.



Rysunek 4.6: Efekt pozostawiony na podłodze po rzuceniu zaklęcia „Wodny Pocisk” przez gracza



Rysunek 4.7: Zaklęcie „Uderzenie Błyskawicy” oraz jego efekt po użyciu czaru

# Rozdział 5

# Specyfikacja wewnętrzna

## 5.1 Przedstawienie idei

Podstawowym założeniem tej pracy inżynierskiej jest sprawienie, aby opisywany system walki był łatwy do zaimplementowania w dowolnej produkcji powstającej na silniku Unity. Z tego powodu należało zadbać o dwie rzeczy. Pierwszą z nich było sprawienie by osoba, która chce wykorzystać ten system, bez poświęcania dużej ilości czasu na zrozumienie implementacji rozwiązania, mogła od razu go wykorzystać. Drugą rzeczą, o którą należało zadbać, było utrzymanie kodu w ten sposób, by dodawanie nowych efektów i zaklęć było dla przyszłego użytkownika intuicyjne i proste.

## 5.2 Opis struktur i najważniejszych klas

Struktury i klasy występujące w projekcie zostały podzielone na podgrupy w zależności od funkcji jaką pełnią w systemie:

- zarządzające życiem i efektami nałożonymi na postacie,
- obsługujące rysowanie symboli w grze i zapisywanie ich w odpowiednim formacie,
- zarządzanie i sterowanie zaklęciami,

- zarządzanie i sterowanie efektami,
- obsługujące poruszanie się gracza wraz z rozpoznawaniem wszystkich stanów postaci,
- komunikowanie się z aplikacją predykcji.

### 5.2.1 Zarządzanie stanem postaci

Na zarządzanie stanem postaci składają się klasy posiadające opisane funkcjonalności:

- **CombatCharacter** - monitorowanie stanu życia postaci, aktywnych efektów magicznych, redukcja lub inkrementacja obrażeń, blokowanie standardowo dostępnych opcji takich jak poruszanie się czy możliwość rzucania zaklęć, zmiany ilości puntów życia postaci, dodawanie efektów zaklęć do listy oraz aktywacja ich koprocedur, a także usuwanie efektów, które zostały nadpisane, ich czas się skończył lub zostały zniesione przez inne zaklęcia.
- **CombatPlayer** - dziedziczy po **CombatCharacter**, pomaga w rzucaniu zaklęć, określając ich kierunek i pozycję startową, nakłada negatywny efekt jeśli symbol został źle odwzorowany.
- **CombatEnemy** - dziedziczy po **CombatCharacter**, przysłania wirtualną funkcję odpowiedzialną za śmierć postaci.

### 5.2.2 Obsługa rysowania symboli

Zarządzaniem rysowaniem symboli zajmują się poniższe klasy o opisanych funkcjonalnościach:

- **Draw** - tworzy pędzel, którym rysowany jest symbol, ustala jego odległość od postaci, co klatkę dorysowuje nowy punkt w pokazanym przez myszkę miejscu na ekranie, zapamiętuje ostatnią lokalizację pędzla.
- **PictureSaver** - zapisuje narysowany symbol w odpowiednim formacie na określonej ścieżce, czyści narysowany pędzlem rysunek, wysyła sygnał, by rozpocząć predykcję za pomocą sieci neuronowej.

### 5.2.3 Zarządzanie zaklęciami

Aby w odpowiedni sposób posegregować zaklęcia dostępne w grze, została stworzona hierarchia dziedziczenia. Korzeniem drzewa jest klasa **Spell**, z której dziedziczą wszystkie zaklęcia. Przechowuje ona informacje o id, nazwie, szukanym celu, typie, zawartym efekcie, a także kto był właścicielem czaru. Posiada też funkcję używaną przez jej dzieci, zwracającą nazwę warstwy, w którą trafiło zaklęcie.

Informacja o celu zaklęcia zapisana jest w zmiennej typu wyliczeniowego **Target** i posiada on poniższe wartości;

- ENEMY - celem zaklęcia ma być przeciwnik rzucającego czar.
- SELF - celem zaklęcia jest osoba rzucająca czar
- TERRAIN - celem zaklęcia jest teren, a nie osoba.
- ALL - celem zaklęcia jest cokolwiek, co jest obiektem.

Informacja o typie zaklęcia zapisana jest w zmiennej typu wyliczeniowego **SpellType** i posiada on poniższe wartości;

- POINT_CLICK - obiekt zaklęcia ma sie pojawić w miejscu, gdzie patrzy się postać w momencie uwolnienia czaru.
- PROJECTILE - obiekt zaklęcia ma się pojawić przy rzucającym czar i przemieszczać się w kierunku, w którym patrzył się w momencie uwolnienia.
- INSTANT - obiekt zaklęcia ma się pojawić niezależnie od pozycji czy kierunku patrzenia rzucającego.

Po **Spell** dziedziczą poniższe klasy, które dzielą zaklęcia pod względem sposobu uwolnienia czaru. Posiadają one określone funkcjonalności:

- **PointClick** - zawiera informacje o wystrzelonym promieniu, który trafił w punkt patrzenia rzucającego, uruchamia stworzenie obiektu zaklęcia na trafionym obiekcie.

- **Projectile** - zawiera informacje o szybkości, zasięgu, a także bezpiecznym czasie wypuszczonego pocisku. Zapamiętuje swoją pozycję startową oraz miejsce zderzenia z innym obiektem. Zarządza stworzeniem reprezentacji pocisku, a także przesuwaniem go w określonym kierunku i czasie. Kontroluje, by na początku wypuszczenia zaklęcia rzucający sam nie został celem czaru.
- **Instant** - wyszukuje rzucającego czar i na nim wywołuje efekt zaklęcia.

Każda z powyższych klas posiada dwie funkcje:

- **OnHit** - wywoływana przez konkretne zaklęcie w momencie trafienia w założony cel.
- **OnVanish** - wywoływana przez konkretne zaklęcie w momencie usuwania swojej reprezentacji z jakiejś przyczyny.

Po **PoinClick**, **Projectile** oraz **Instant** dziedziczą kolejne klasy będące liśćmi tej hierarchii. Każda z poniższych klas implementuje jedno konkretne zaklęcie i zarządza, co się dzieje po trafieniu czaru w określony cel. Poniżej opisane zostały te konkretne zaklęcia i ich funkcjonalności:

- **AirBurst** - należy do grupy „PointClick". Niezależnie od tego, czy zaklęcie trafiło w warstwę „Player", „Enemy" czy „Ground" ma zostać wywołany efekt przypisany do czaru.
- **BlackHole** - należy do grupy „PointClick". Jeśli trafionym przez zaklęcie obiektem była postać, to efekt czaru ma się wywołać na podłodze, na której stał cel. W innym przypadku efekt czaru ma się pojawić dokładnie w miejscu kolizji.
- **BlinkOfLightning** - należy do grupy „PointClick". Jeśli celem zaklęcia była warstwa „Ground", to rzucający zaklęcie zostanie przeteleportowany w miejsce trafienia. Gdy trafiony czarem zostanie obiekt warstwy „Player" lub „Enemy" to nastąpi zamiana pozycji rzucającego zaklęcie z ofiarą.
- **EarthPrison** - należy do grupy „Projectile". Po sprawdzeniu, czy nie nastąpiło samo trafienie rzucającego, w momencie wejścia w kolizję z warstwą

„Enemy" lub „Player" zostanie wywołany efekt czaru. Po zetknięciu z warstwą „Ground" nic się nie stanie i reprezentacja będzie się przemieszczać wzdłuż płaszczyzny otoczenia.

- **Fireball** - należy do grupy „Projectile". Po sprawdzeniu, czy nie nastąpiło samo trafienie rzucającego, w momencie wejścia w kolizję z warstwą „Enemy" lub „Player" zostanie wywołany efekt czaru. Po zetknięciu z warstwą „Ground" lub inną nastąpi zniszczenie reprezentacji zaklęcia.

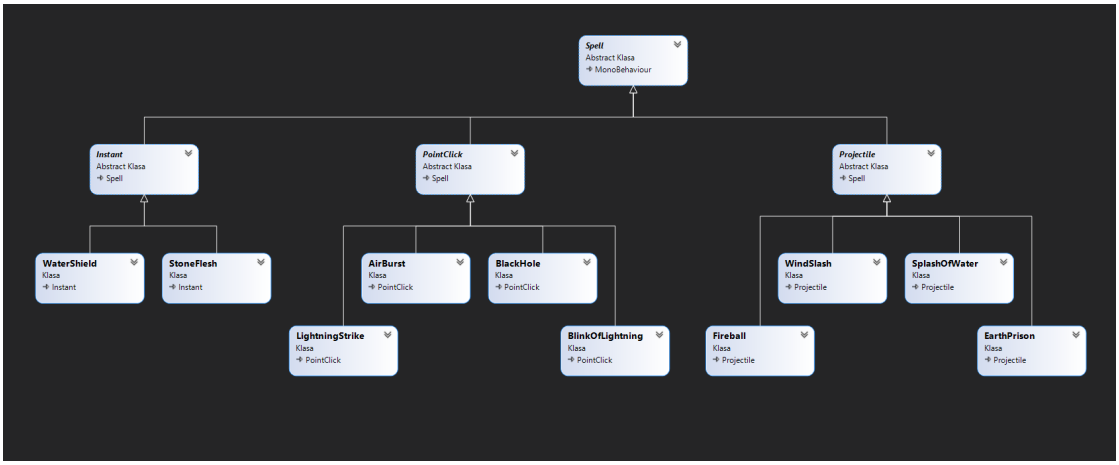
- **LightningStrike** - należy do grupy „PointClick". W momencie wejścia w kolizję z warstwą „Enemy" lub „Player" zostanie wywołany efekt czaru. Niezależnie od celu trafienia na chwilę pojawi się spadający z nieba, imitujące błyskawicę obiekt.

- **SplashOfWater** - należy do grupy „Projectile" Po sprawdzeniu, czy nie nastąpiło samo trafienie rzucającego, w momencie wejścia w kolizję z warstwą „Enemy" lub „Player" zostanie wywołany efekt czaru. W przypadku trafienia w warstwę „Ground" zostanie wywołana alternatywa efektu.

- **StoneFlash** - należy do grupy „Instant". Zaraz po użyciu tego zaklęcia na rzucającym zostanie wywołany efekt czaru, który zwiększy odporność gracza na wszytskie obrażenia.

- **WaterShield** - należy do grupy „Instant". Zaraz po użyciu tego zaklęcia na rzucającym zostanie wywołany efekt czaru, który otoczy gracza wodną tarczą. Ta zasłona umożliwi mu uchronienie się przed niektórymi zaklęciami przeciwnika.

- **WindSlash** - należy do grupy „Projectile". Po sprawdzeniu, czy nie nastąpiło samo trafienie rzucającego, w momencie wejścia w kolizję z warstwą „Enemy" lub „Player" zostanie wywołany efekt czaru. Ten czar nie reaguje na kolizję z warstwą „Ground", gdy wejdzie z nią w reakcje, to przeniknie przez nią.




Rysunek 5.1: Diagram UML klas zaklęć w sytemie

### 5.2.4 Zarządzanie efektami

Wszystkie efekty dostępne w systemie dziedziczą z jednej klasy o nazwie **Effect**. Zawiera ona informacje o nazwie, elemencie, czasie trwania, częstotliwości powtarzania i czasie startu efektu. Nadpisuje czasu startowy, jeśli dany efekt został ponownie dodany.

Element, czyli typ efektu pomaga dodawać relacje między efektami i zachować w nich porządek. Jest określany za pomocą zmiennej typu wyliczeniowego **Element**, a przyjmuje wartości:

- FIRE - typ ognia
- WATER - typ wody
- EARTH - typ ziemi
- LIGHTNING - typ błyskawic
- WIND - typ wiatru

Klasa **Effect** posiada dwie funkcje abstrakcyjne:

- **ReleaseEffect** - ta funkcja jest wywoływana przez zaklęcia w momencie zetknięcia z właściwym celem. Tu jest zaimplementowany rodzaj zdarzenia, jaki ma się wykonać jednokrotnie w danym efekcie — tylko przy trafieniu.

Do takich operacji zalicza się np. ustawianie rodzica reprezentacji lub pozycji efektu.

- **EffectCoroutine** - w tej koprocedurze odbywają się zdarzenia, jakie ma wywołać efekt po czasie lub powtarzane cyklicznie przez pewien okres trwania. Wywoływana jest wewnątrz **ReleaseEffect**.

Wszystkie efekty dziedziczące z klasy **Effect** zostały opisane poniżej:

- **Ablaze** - efekt typu ogniowego. Uruchamia się w momencie, gdy cel, który posiada na sobie efekt „Burning", zostanie trafiony zaklęciem „WindSlash". Jego działanie polega na zadawaniu obrażeń co jakiś czas, aż do momentu wygaśnięcia. Działa jak wzmocniona wersja „Burning".
- **AirPlatform** - efekt typu wietrznego. Uruchamia się, gdy postać zostanie wyrzucona przez tubę powietrzną z efektu „AirPopup", tworząc platformę, na której postać może się utrzymać i poruszać.
- **AirPopup** - efekt typu wietrznego. Uruchamia się bezpośrednio po odpowiednim zaklęciu i tworzy tubę powietrzną, która wypycha postaci i niektóre zaklęcia typu „Projectile" w górę.
- **BlackGravity** - efekt typu ziemnego. Uruchamia się bezpośrednio po użyciu zaklęcia „BlackHole". Przez cały czas trwania ściąga wszystkie postacie i zaklęcia typu „Projectile", które znalazły się w zasięgu siły, do swojego środka — jak czarna dziura.
- **Burning** - efekt typu ogniowego. Uruchamia się, gdy cel zostanie trafiony zaklęciem „Fireball" i zadaje obrażenia co jakiś czas przez cały okres trwania.
- **Electrocuted** - efekt typu błyskawic. Uruchamia się, jeśli osoba posiadająca na sobie efekt typu wodnego, zostanie trafiona zaklęciem elektrycznym, takim jak „LightningStrike". Postać z nałożonym takim statusem nie jest w stanie rzucać zaklęć przez cały okres trwania.
- **Heal** - efekt typu ziemnego. Uruchamia się, gdy postać posiadająca na sobie status „StoneProtection" zostanie zmoczona wodą — czyli dostanie efekt „Wet". Wtedy trafiony tą serią czarów będzie się leczył przez określony czas.

- **Imprisonment** - efekt typu ziemnego. Uruchamia się bezpośrednio po trafieniu zaklęciem „EarthPrison". Trafiony cel zostaje obudowany z każdej strony blokami kamienia, tak by nie mógł uciec, a nadal był odsłonięty na ostrzał przeciwnika. Po określonym czasie efekt znika.

- **Puddle** - efekt typu wodnego. Uruchamia się w momencie, gdy wystrzelona wodna kula z zaklęcia „SplashOfWater" trafi w warstwę „Ground". Wynikiem efektu jest powstanie na powierzchni rozlanej kałuży, która zostanie na swoim miejscu przez pewien czas. Gdy jakaś postać wejdzie w plamę wody, to zostanie na nią nałożony efekt „Wet".

- **Shock** - efekt typu elektrycznego. Uruchamia się zaraz po trafieniu zaklęcia „LightningStrike" i sprawia, że przez określony czas cel nie jest w stanie się poruszyć.

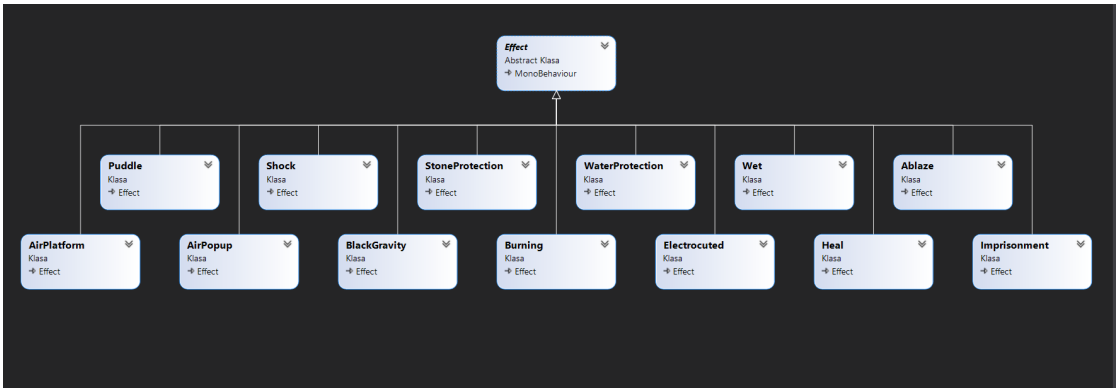
- **StoneProtection** - efekt typu ziemnego. Uruchamia się zaraz po rzuceniu zaklęcia „StoneFlash" i zwiększa odporność na obrażenia osoby, która rzucała czar.

- **WaterProtection** - efekt typu wodnego. Uruchamia się zaraz po użyciu zaklęcia „WaterShield" i sprawia, że wokół rzucającego na pewien czas roztacza się wodna bańka. Ochrona ta niweluje ofensywne efekty wszystkich elementów poza błyskawicami. W przypadku zetknięcia z ziemnym elementem tarcza sama zostaje zniszczona, a w przypadku błyskawic ochrona pozostaje, lecz są nakładane niechciane efekty porażenia.

- **Wet** - efekt typu wodnego. Uruchamia się, gdy postać zostaje trafiona zaklęciem „SplashOfWater" lub sama wejdzie w obszar działania „Puddle". Efekt ten wchodzi w wiele reakcji z innymi. Potrafi gasić elementy ogniowe, być powodem dodatkowych porażeń od błyskawic, a nawet wywołać leczenie.

### 5.2.5 Obsługa poruszania się gracza

Obsługą poruszania się gracza zajmują się poniższe klasy, które zawierają określone funkcjonalności:

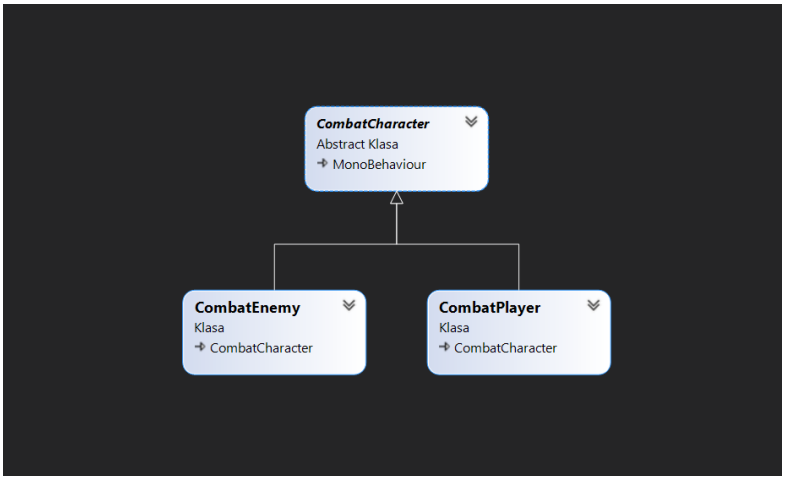



Rysunek 5.2: Diagram UML klas efektów w sytemie

- **PlayerMovement** - zarządza przemieszczaniem się gracza po świecie, normalizuje kierunek ruchu, zbiera informacje o stanie klawiatury i podejmuje odpowiednie reakcje w zależności od tego, który klawisz został wciśnięty, zmienia i monitoruje stany postaci gracza, analizuje otoczenie, w jakim znajduje się gracz, np. jest w powietrzu, jest na pochylni lub jest na ziemi. Skrypt zmienia i monitoruje prędkość ruchu gracza, jego prędkość upadku, a także opory powietrza, jakie na niego działają.
- **PlayerLook** - zarządza rotacją kamery, reguluje czułość obrotu, wprowadza ograniczenia kątów rotacji kamery.
- **MoveCamera** - przemieszcza kamerę co klatkę, na pozycję znajdującą się na szczycie reprezentacji gracza na planszy.

### 5.2.6 Komunikacja z siecią neuronową

Komunikacją z siecią neuronową z poziomu Unity zajmuje się skrypt **NetworkManager**. Przy starcie aplikacji przygotowuje on folder, w którym za pomocą pliku tekstowego będą przekazywane informacje z programu predykcji za pomocą sieci neuronowej o wynikach. Skrypt ten będzie później udostępniał pozostałym menadżerom odczytane rezultaty. Gdy zostanie narysowany i zapisany symbol, włącza się funkcja **PrepareProc**, która wykorzystując bibliotekę **System.Diagnostics** uruchamia zewnętrzny proces aplikacji predykcji sieci neurono-




Rysunek 5.3: Diagram UML klas postaci walczących w systemie

wej, przekazuje mu jako argument ścieżkę do folderu, gdzie mają zostać zapisane wyniki, a następnie w asynchroniczny sposób oczekuje na zakończenie procesu. Gdy ten dobiegnie końca, odczytuje z pliku wyniki testu i uruchamia wywołanie zaklęcia o przewidywanym indeksie.

## 5.3 Wykorzystane komponenty, moduły oraz biblioteki

1. **System.Diagnostics**
   To przestrzeń nazw zawierająca klasy pozwalające na współdziałanie z zewnętrznymi procesami systemowymi oraz ich analizę. W systemie posłużono się tą biblioteką, by w czasie trwania programu uruchamiać zewnętrzną aplikację predykcji za pomocą sieci neuronowej. Takie podejście było konieczne z powodu braku integracji potrzebnych bibliotek .NET ze środowiskiem Unity, co uniemożliwiało napisanie wewnętrznego skryptu do predykcji wyników.

2. **ML.NET**
   To biblioteka stosowana do dodawania uczenia maszynowego do aplikacji .NET. Pozwala ona na tworzenie scenariuszy, gdzie można wybrać odpowiedni algorytm i dane wejściowe, aby zaprojektować potrzebny model. Po


Rysunek 5.4: Diagram UML pozostałych klas, interfejsów i typów wyliczeniowych w systemie

wytrenowaniu gotowy model może zostać dodany do aplikacji, by tworzyć predykcje rysowanych obrazów. W ramach projektu napisana została niezależna aplikacja, w której sieć została wytrenowana i przetestowana, a następnie utworzono drugi program, który ma wykonywać predykcje i zapisywać wyniki do pliku, tak aby dało się je odczytać z poziomu aplikacji korzystającej z silnika Unity

3. **ML.NET Model Builder**
   ML.NET Model Builder to graficzne rozszerzenie do Visual Studio. Za jego pomocą można łatwo zaprojektować i wytrenować modele uczenia maszynowego takich jak interpretacje danych rynkowych czy choćby rozpoznawanie elementów obrazów. Model Builder stosuje AutoML (automatyczne uczenie maszynowe), co umożliwia stosowanie różnych algorytmów i ustawień, dzięki którym można drogą testowania znaleźć takie, które najbardziej odpowiadają wybranemu scenariuszowi. W projekcie posłużono się tym rozwiązaniem, aby dobrać jak najlepszy algorytm uczący strukturę sieć, a także, by stworzyć odpowiednie, łatwe do obsługi struktury wejściowe i wyjściowe.

4. **Rigidbody**
   Rigidbody to komponent silnika graficznego Unity wprowadzająca fizyczne symulacje. Po dodaniu komponentu Rigidbody do obiektu w grze możliwe jest kontrolowanie silnika fizycznego dla tej jednostki. Jedną z dostępnych opcji jest nakładanie sił na ciała w zbliżony do rzeczywistości sposób. Poza tym możliwe jest wpływanie na grawitację czy opory ruchu. W systemie zdecydowano się na użycie tego komponentu, by zaimplementować ruch postaci. Takie rozwiązanie ułatwiło późniejsze tworzenie efektów zaklęć, takich jak czarna dziura przyciągająca wszystkie fizyczne obiekty do środka, eksplozje po wybuchu lub odepchnięcia wiatrem.

## 5.4 Zastosowane wzorce projektowe

W systemie użyto wzorca projektowego Singleton, który umożliwia tworzenie tylko jednego obiektu danej klasy w instancji zapewniającego dostęp globalny do metod i atrybutów tego obiektu. Jest to bardzo często używany wzorzec przez

programistów Unity [2] ze względu na prostotę jego integracji z systemem gdzie strukturą są skrypty. W przypadku tworzonego systemu posłużenie się tym rozwiązaniem jest bardzo wskazane, gdyż w trakcie gry może istnieć tylko jeden gracz, który będzie rysował symbole wywołujące rzucanie zaklęć na podstawie predykcji za pomocą sieci neuronowej. W projekcie wszystkie klasy należące do wzorca Singleton posiadają przeznaczone zadania, dzięki czemu zarządzanie nimi w projekcie jest intuicyjne i łatwe do zrozumienia przez potencjalną osobę, która wykorzysta ten system walki.

# Rozdział 6

# Weryfikacja i walidacja

## 6.1 Sposób testowania systemu

Kolejne funkcjonalności były implementowane i wielokrotnie testowane:

1. W pierwszej kolejności przetestowano sprawność wytrenowanego modelu sieci neuronowej. Zrobiono to, wykonując kilka testowych klasyfikacji w tym samym programie, w którym sieć była szkolona, na koniec sekwencji trenowania. Później stworzono osobną aplikację, gdzie model najpierw był wczytywany, a później zwracał predykcję wyliczoną na określonym obrazie.

2. Drugą testowaną rzeczą był mechanizm przemieszczania postaci. Sprawdzano, jak reprezentacja gracza zachowuje się przy ścianach, na krawędziach stopni, lub urwisk, a także na pochylniach o różnych stopniach nachylenia.

3. Kolejną testowaną mechaniką było rysowanie i zapis symboli. Pierwszym krokiem było utworzenie sceny tylko i wyłącznie w 2D, z nieruchomą kamerą i napisanie skryptu, który będzie w stanie narysować na płaskiej powierzchni rysunek, a następnie go zapisać. Gdy ten krok został wykonany, należało przenieść mechanizm do przestrzeni 3D i podłączyć go do ruchomej kamery.

4. Następnie przetestowano uruchamianie zewnętrznego procesu z poziomu edytora Unity. Najpierw testowanie zostało wykonane na prostych, aplikacjach wydanych przez znanych wydawców takich jak Notepad czy Paint, by unik-

nąć problemów z antywirusem, a później na własnej aplikacji z podpunktu pierwszego.

5. Kolejnym krokiem testowania systemu było sprawdzenie działania wszystkich stworzonych zaklęć i efektów. Ważnym elementem było przejrzenie czy wszystkie relacje między czarami następują tak, jak powinny, a ich pozostałości są usuwane w odpowiednich momentach.

6. Ostatnim elementem testowania było sprawdzenie, czy całość systemu nadal będzie funkcjonować tak jak w edytorze, po wygenerowaniu programu wynikowego, oraz jak sprawdzi się, gdy zostanie on zaimportowany do nowo utworzonego projektu.

Po każdym z wymienionych etapów następował proces testowania całości aplikacji, a zauważone błędy były eliminowane tak, by nie występowały później w procesie tworzenia. Niektóre błędy zostały zauważone dopiero przy ostatnim etapie, jakim było wygenerowanie programu wynikowego co sprawiło, że największa ilość znalezionych błędów wystąpiła właśnie wtedy.

System został przetestowany przez twórcę oraz przez osoby niezwiązane z projektem, a wszystkie znalezione błędy zostały wzięte pod uwagę i wyeliminowane.

## 6.2 Wykryte i usunięte błędy

Jednym z najważniejszych błędów, jakie zostały wykryte przy procesie tworzenia była obsługa zewnętrznego procesu, który w edytorze nie miał żadnych problemów z uruchamianiem, natomiast w programie wynikowym, zamykał się zaraz po starcie. Okazało się, że problem ten wynikł z wybranego w projekcie zestawu narzędzi, którego domyślna wersja **IL2CPP** nie obsługuje klasy **Process** z biblioteki **System.Diagnostics**. Naprawienie tego błędu było bardzo proste, wystarczyło zmienić zestaw narzędzi na wersję **Mono** i całość aplikacji działała jak w edytorze, ale przysporzyło to sporo problemów, z powodu braku możliwości wyszukiwania błędów w programie wynikowym programu.

Kolejny wykryty błąd, którego znalezienie zajęło dużo czasu, miał miejsce przy wywoływaniu klasyfikacji narysowanych wewnątrz Unity obrazów. Model bez pro-

blemu klasyfikował zdjęcia szkicowane w programach takich jak Paint, natomiast te, z aplikacji zawsze klasyfikował jako jedną cyfrę, niezależnie od tego, co się naszkicowało. Problem polegał na tym, że domyślne tło w Unity nie jest perfekcyjnie białe, a lekko niebieskie. Program predykcji normalizował obraz i widział wszystkie kolorowe piksele jak czarne, więc otrzymany przez niego obraz był całkowicie ciemny. Naprawienie tego błędu było również proste, należało zmienić kolor tła kamery renderującej na biały.

Podczas testowania zaklęć został wykryty błąd związany z różnym oddziaływaniem sił w zależności od tego, czy laptop, na którym wykonywany był projekt, był podpięty do zasilania, czy nie. Błąd wynikał z tego, że nakładane na ciała siły z komponentu **Rigidbody** są zależne od zmiany czasu. Laptop działający na zasilaniu bateryjnym ograniczał moc procesora, by zyskać na żywotności, natomiast będąc podpięty do sieci już nie. Ta różnica sprawiała, że każda nakładana siła funkcjonowała z inną mocą. Naprawienie tego błędu ograniczyło się do przemnożenia wartości nakładane siły przez **Time.deltaTime** - czyli wartość w sekundach, jaka upłynęła od ostatniej klatki, do aktualnej.

# Rozdział 7

# Podsumowanie i wnioski

Cel pracy, jakim było zaprojektowanie systemu walki do gry cRPG opartego na rysowaniu symboli, został w pełni zrealizowany. Założenia funkcjonalne i niefunkcjonalne zostały spełnione tak samo jak cele projektu.

Dalsze prace przy rozwoju systemu można podjąć w kilku kierunkach. Jednym z nich jest dalsze rozwijanie go o następne zaklęcia, aby zwiększyć możliwości gracza podczas potyczki z przeciwnikiem. Inną możliwością jest stworzenie dodatkowych połączeń pomiędzy efektami, aby pozwolić na tworzenie jak najsilniejszych kombinacji w walce. Wszystkie dostępne mechaniki można by opatrzyć dużo lepszą reprezentacją graficzną wykorzystującą modele i zaawansowane systemy cieniowania. Ostatnim elementem rozwoju w przyszłości mogłoby być przełożenie zewnętrznej aplikacji predykcji sieci neuronowej na wewnętrzny skrypt, co z pewnością zoptymalizowałoby grę.

W czasie pracy nad systemem napotkano problemy związane z brakiem wsparcia niektórych bibliotek przez środowisko Unity. Opinią autora jest, że, pomimo że developerzy często aktualizują i pracują nad silnikiem od bardzo dawna, to często zostawiają niektóre rozwiązania niekompletne, lub niepotrzebnie skomplikowane co nieraz może przysporzyć więcej problemów niż korzyści.

Dzięki temu projektowi możliwe było wykorzystanie wiedzę zdobytą na studiach w praktyce. Wiele z założenia prostych zasad zaczyna być skomplikowanych, gdy rośnie rozmiar aplikacji. Utrzymanie czystości kodu jest w takich projektach kluczowe, gdyż nieraz wraca się do dawno napisanych fragmentów i chce się poświę-

cić jak najmniej czasu na zrozumienie sposobu działania danej części programu. Jednak udało się stworzyć zaawansowany system walki, który może być z łatwością rozwijany i wdrążony w dowolnym projekcie Unity.

# Bibliografia


[1] Expert Ceneo. Ranking gier. top najlepszych gier 2021 na pc, ps, xbox i planszówek.
`https://ekspert.ceneo.pl/ranking-gier#1`. [**dostęp z dnia: 2021-11-24**].

[2] Joseph Hocking. *Unity in Action: Multiplatform Game Developement in C#*. HELION, ul.Kościuszki 1c, 44-100 Gliwice, 2017.

[3] Mahnoor Javed. The best machine learning algorithm for handwritten digits recognition.
`https://towardsdatascience.com/the-best-machine-learning-algorithm-for-handwritten`

[**dostęp z dnia: 2021-11-29**].

[4] SD Kulik. Neural network model of artificial intelligence for handwriting recognition. *Journal of Theoretical & Applied Information Technology*, 73(2), 2015.

[5] S Lambottin. The fundamental pillars of a combat system, 2012.

[6] Yann LeCun, LD Jackel, Leon Bottou, A Brunot, Corinna Cortes, John Denker, Harris Drucker, Isabelle Guyon, UA Muller, Eduard Sackinger, i in. Comparison of learning algorithms for handwritten digit recognition. *International conference on artificial neural networks*, wolumen 60, strony 53–60. Perth, Australia, 1995.

[7] Nick Montfort. Combat in context. *Game Studies*, 6(1):1, 2006.

[8] Christopher Dristig Stenström, Staffan Björk. Understanding computer role-playing games. 2008.

[9] WePC. Video game industry statistics, trends and data in 2021. `https://www.wepc.com/news/video-game-statistics/`. [**dostęp z dnia: 2021-11-25**].

# Dodatki

# Spis skrótów i symboli

cRPG komputerowa gra fabularna (ang. *computer Role Playing Game*)

acRPG komputerowa gra fabularna akcji (ang. *action computer Role Playing Game*)

CBR cRPGs gry fabularna o ciągłym ryzyku walki (ang. *Constant Battle Risc cRPGs*)

F1-Score średnia harmoniczna pomiędzy precyzją (precision) i czułością (recall). Przyjmuje wartości od 0 do 1. Im precyzyjniejsz jest algorytm klasyfikujący tym wyższy wynik. Kiedy czułość i precyzja są idealne to przyjmuje wartośc 1.

SVM algorytm klasyfikacyjny - Wsparcie Maszyn Wektorowych (ang. *Support Vector Machines*)

KNN algorytm klasyfikacyjny - K Najbliższych Sąsiadów (ang. *K Nearest Neighbours*)

KOTOR Gwiezdne Wojny: Rycerze Staarej Republiki (ang. *Star Wars: Knights of The Old Republic*)

zasóby użyte w projekcie materiały o różnym charakterze. Mogą to być między innymi tekstury, modele czy nawet gotowe skrypty (ang. *assets*)

GUI graficzny interfejs użytkownika (ang. *grafical user interface*)

# Lista dodatkowych plików, uzupełniających tekst pracy

W systemie do pracy dołączono dodatkowe pliki zawierające:

- źródła aplikacji testowej systemu walki,
- źródła aplikacji predykcji za pomocą sieci neuronowej,
- źródła aplikacji wytrenowania i przetestowania sieci neuronowej,
- program wynikowy aplikacji testowej systemu walki.

# Spis rysunków