

**WYDZIAŁ**

**TRANSPORTU, ELEKTROTECHNIKI**

**I INFORMATYKI**

Kierunek: …………………….......…........…....

w zakresie: …………………….......……….......

**PRACA DYPLOMOWA INŻYNIERSKA**

**TYTUŁ PRACY**

**Gra platformowa z elementami sztucznej inteligencji wykonana z użyciem silnika UNITY**

| Wykonał(a): | Promotor: |
| --- | --- |
| ……………………………………….. | ………………………………………. |
| nr albumu ……….. |  |
| *Praca przyjęta do obrony:*  *…………………………………………*  *data i podpis promotora* | |

Radom 2023

1. Sztuczna inteligencja

1.1. Sztuczna inteligencja w grach

1.2. Czat GPT

1.3. Rozwój sztucznej inteligencji

1. Projekt i implementacja

2.1. Rozgrywka

2.2. Interfejs użytkownika

2.3. Animacja

2.4. Wykorzystane technologie

2.4.1. Unity

2.4.2. VisualStudio

2.4.3. Git/TortoiseGit/GitHub

1. Zastosowanie sztucznej inteligencji w grze

3.1. Pathfinding

3.2. Pole widzenia

3.3. Maszyna stanów skończonych

3.4. Uczenie przez wzmacnianie

**Streszczenie**

W ramach wykonania projektu została wykonana gra platformowa w silniku Unity. Zawiera ona elementy sztucznej inteligencji. Praca składa się z trzech rozdziałów w których omawiane są poszczególne etapy tworzenia gry. Pierwszy dotyczy sztucznej inteligencji. Rozdział ten ma za zadanie krótko zaznajomić z samym pojęciem sztucznej inteligencji, zastosowaniem jej oraz zawiera przykład zwycięstwa uczenia przez wzmacnianie nad innymi podejściami. Następny rozdział dotyczy tylko tej gry. Przedstawiony jest w niej sposób w jaki została wykonana, zawiera opis rozgrywki, a w nim implementacja sterowania postacią. Gra zawiera interfejs użytkownika, wykonany tak aby był intuicyjny dla przeciętnego odbiorcy. Ostatnim poruszonym tematem tego rozdziału są wykorzystane technologie podczas pracy nad projektem. Ostatni rozdział dotyczy wykorzystania sztucznej inteligencji w grze.

**Wstęp**

Moja praca inżynierska składa się z aplikacji, która zajęła większość czasu przeznaczonego na pracę dyplomową oraz niniejsza praca stanowiąca opis produktu. Celem pracy było stworzenie gry w środowisku Unity. Gra z założenia miała być prosta w odbiorze, stworzona w czytelny sposób, możliwa do rozbudowania w przyszłości. Poszczególne skrypty w miarę możliwości były pisane w taki sposób aby można je było wykorzystać wielokrotnie.

Gra RobotGame jest to platformowa gra komputerowa, w której gracz wciela się w postać robota. Zadaniem gracza jest dojście do wyznaczonego miejsca poprzez pokonywanie przeszkód. W tym celu może strzelać, rzucać, zwiększać tempo poruszania się jednak jest w tym ograniczony. W związku z tym należy używać ulepszeń tylko wtedy kiedy to konieczne. W zamyśle gra miała być łatwa do przejścia, zazwyczaj wymagając minimum myślenia czy precyzji. Nie posiada też instrukcji poza sterowaniem w ustawieniach. Celem było stworzenie jej w taki sposób, aby stopniowo, w łatwy sposób można było się zaznajomić z panującymi tam zasadami.

Aplikacja stworzona jest w większości przy użyciu tradycyjnych technik programowania.

Istotnym elementem w pracy jest sztuczna inteligencja. Powodem jej zastosowania była chęć poznania jej możliwości. Zawarte elementy SI w pracy mają pokazać sens i korzyści płynące z używania jej w grach wideo.

Niniejsza praca jest opisem produktu i zastosowanych w niej technik. Rozpoczyna się od rozdziału opisującego tematykę projektu. Rozdział ten ma za zadanie krótko zaznajomić z samym pojęciem SI, omawia jej zastosowanie oraz zawiera przykład zwycięstwa uczenia przez wzmacnianie nad innymi podejściami. Następny rozdział dotyczy tylko tej gry. Przedstawiony jest w niej sposób w jaki została wykonana, zawiera opis rozgrywki, a w nim implementacja sterowania postacią. Gra zawiera interfejs użytkownika, wykonany tak aby był intuicyjny dla przeciętnego odbiorcy. Ostatnim poruszonym tematem tego rozdziału są wykorzystane technologie podczas pracy nad projektem. Narzędzia wykorzystane były niezbędne podczas pracy lub znacząco ułatwiały ją. Ostatni rozdział dotyczy wykorzystania sztucznej inteligencji. Zaczyna się on od opisania wykorzystania algorytmu *pathfinding*. Opisany jest tu też mechanizm działania maszyny stanów skończonych jako sposobu implementacji sztucznej inteligencji oraz zawartego w niej pola widzenia. Na końcu została opisana rzecz, na którą zostało poświęcone zdecydowanie najwięcej czasu i która sprawiła najwięcej trudności. Mowa tutaj o uczeniu przez wzmacnianie, inaczej Reinforcement Learning. Samo pojęcie RL zostało opisane w pierwszym rozdziale. Celem tego podejścia było zobrazowanie działania uczenia maszynowego. Podczas gdy reszcie elementów w grze zostały z góry przypisane zasady w jaki mają funkcjonować, tak tutaj agent,bo tak się go nazywa w środowisku ml-agents musiał się sam nauczyć jak rozwiązać postawione przed nim zadanie. Celem tego było pokazanie różnicy między tym podejściem a zwykłym programowaniem, jego zalet ale też i wad oraz wyciągnięcie wniosku do jakich zadań najlepiej byłoby wykorzystać to podejście w przyszłości.

**1. Sztuczna inteligencja**

**1.1. Sztuczna inteligencja w grach**

Głównym celem stosowania w grach wideo algorytmów sztucznej inteligencji jest jak najlepsze odwzorowanie ludzkiego zachowania czy inteligentne zachowanie przedmiotów nieożywionych. Założenie to wydaje się dobre, ponieważ człowiek ma skłonność do personifikowania obiektów, z którymi wchodzi w interakcje.

Jednym z obszarów sztucznej inteligencji jest uczenie maszynowe, inaczej machine learning(ML). Algorytmy ML poprawiają się automatycznie poprzez doświadczenie. Polega to na karmieniu systemu informacjami i obserwacjami, które można wykorzystać do znalezienia wzorców i przewidywania przyszłych wyników. System musi nauczyć się pożądanego mapowania wejścia-wyjścia. W ten sposób będzie mógł wybrać najlepsze działanie do wykonania w celu optymalizacji jego wyniku [2].

Algorytmy uczenia maszynowego są jednym z najczęściej wykorzystywanych narzędzi sztucznej inteligencji w grach. ”Uczenie maszynowe jest zbiorem technik, których można użyć do przetwarzania dużych ilości danych w najefektywniejszy sposób i które pozwolą nam uzyskać z tych danych ważne wyniki i przeprowadzić analizę tych danych”[**numberUM**]

Innym obiecującym obszarem sztucznej inteligencji są sieci neuronowe, które przetwarzają informacje i uczą się na podstawie doświadczeń. Współczesne sieci neuronowe są zwykle oparte na tzw. sieciach głębokich, czyli sieciach neuronowych, które składają się z wielu warstw. W każdej warstwie znajdują się neurony, które przetwarzają informacje i przekazują je do kolejnej warstwy. „Sieci neuronowe są podstawą głębokiego uczenia się i są kluczowym elementem, który pozwala nam uczyć się przydatnych reprezentacji środowiska w uczeniu się przez wzmacnianie”.[**numberSN**].

Sieci neuronowe można trenować za pomocą algorytmów uczenia maszynowego. W pracy zostało wykorzystane uczenie się przez wzmacnianie (Reinforcement Learning, RL) do trenowania sieci neuronowej. Uczenie przez wzmacnianiem polega na uczeniu agenta podejmowania decyzji poprzez interakcję z dynamicznym otoczeniem w celu maksymalizacji długoterminowej nagrody. Metoda ta polega na powiedzeniu systemowi co jest dobre, a co złe. System wykonuje losowe działania i w zależności od tego czy są one dobre czy złe, przyznajemy nagrodę lub karę. Ponieważ jego zadaniem jest maksymalizacja nagrody, system w końcu zrozumie jakie działania ma podejmować aby nagroda była jak najwyższa.

“w uczeniu przez wzmacnianie nie ma odpowiedzi, ale agent wzmacniający decyduje, co zrobić, aby wykonać dane zadanie. W przypadku braku zbioru danych szkoleniowych jest zobowiązany do wyciągnięcia wniosków ze swojego doświadczenia.”[**numberUPW**]

**1.2 Chat GPT**

Dobrym przykładem wykorzystania sztucznej inteligencji jest Chat GPT. To duży model językowy stworzony przez firmę OpenAI. Jednym z najważniejszych algorytmów, które zostały wykorzystane do jego treningu, są sieci neuronowe. W tym przypadku, sieci neuronowe służą do przetwarzania i analizowania dużych ilości tekstów, ucząc się ich gramatyki i kontekstu, co pozwala generować odpowiedzi na różne pytania.

Kolejnym algorytmem, który został wykorzystany podczas treningu Chat-u GPT, jest algorytm oparty na Transformer. Transformer to nowoczesny model językowy, który umożliwia zrozumienie złożonych kontekstów w tekście i wykorzystanie tej wiedzy do generowania trafnych i precyzyjnych odpowiedzi. Transformer został stworzony w celu usprawnienia przetwarzania długich sekwencji danych, co jest problematyczne dla tradycyjnych sieci neuronowych.

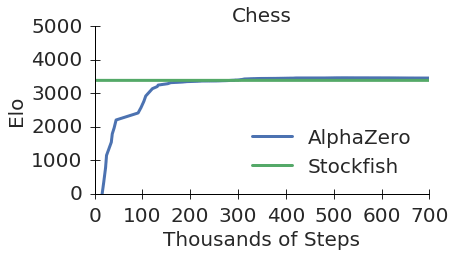
Do wyszkolenia modelu językowego wykorzystano również techniki uczenia nienadzorowanego, takie jak uczenie się kontrastywne i klasyfikacja bez nadzoru, które pozwalają na analizowanie i przetwarzanie danych bez wykorzystania etykiet lub danych uczących.

W przypadku gier, Chat GPT może zostać wykorzystany do generowania tekstu dialogów, misji i zadań dla graczy. Na przykład, w grach RPG, Chat GPT może stworzyć interaktywne postacie, które będą reagować na wybory gracza, dzięki czemu gracz może czuć się bardziej zanurzony w fabule gry. Chat GPT może również pomóc w tworzeniu botów do gier, które będą miały zdolność do rozmowy i interakcji z graczami. Dzięki temu boty mogą udzielać porad i odpowiedzi na pytania graczy, a także zapewniać interaktywną i spersonalizowaną grę.

**1.3. Rozwój sztucznej inteligencji**

W ciągu ostatnich lat rozwój and sztuczną inteligencją znacznie przyspieszył. Umożliwił to m.in. dużo lepszy sprzęt niż np. w latach 90. ubiegłego wieku. Pojęcie sztucznej inteligencji nie jest jednak niczym nowym. Rozwój sztucznej inteligencji doskonale obrazują gry strategiczne takie jak szachy. Jeszcze pół wieku temu mogłoby się wydawać, że komputer nie jest w stanie pokonać człowieka w grze, w którą ludzkość gra od setek lat. Wkrótce ta bariera mentalna została przełamana. “Informatycy przez dziesięciolecia postrzegali szachy jako miernik dla sztucznej inteligencji. Kalkulatory gry w szachy pojawiły się pod koniec 1970 roku, ale minęła kolejna dekada, zanim zespół absolwentów Carnegie Mellon University zbudował [pierwszy komputer - zwany Deep Thought - aby pokonać wielkiego mistrza](https://www.aaai.org/ojs/index.php/aimagazine/article/viewFile/753/671) w zwykłej grze turniejowej” [3] . Prawdziwy przełom nastąpił jednak w 1996 roku gdy Deep Blue (Rys. 2.) - komputer grający w szachy firmy IBM wygrał jeden z sześciu meczów z ówczesnym mistrzem świata - Garrim Kasparowem , a już w 1997 roku wygrał z nim 3.5-2.5.

Z biegiem lat pojedynki człowieka z silnikami szachowymi miały coraz mniejszy sens, ponieważ człowiek był coraz bardziej bezradny w starciu z maszyną. W grudniu 2017 roku nastąpił kolejny przełom. Opracowany przez firmę zajmującą się sztuczną inteligencją DeepMind, przejętą przez Google program komputerowy AlphaZero pokonał, a raczej zdeklasował dotychczasowego lidera wśród silników szachowych - Stockfisha. Spośród 100 rozegranych partii AlphaZero wygrał 28 i nie przegrał żadnej. “Jest to program komputerowy, który osiągnął praktycznie niewyobrażalny poziom zabawy, używając tylko [uczenia się przez wzmacnianie](https://www.chess.com/news/view/david-silver-alphazero-reinforcement-learning) i samodzielnej gry w celu trenowania swoich sieci neuronowych. Innymi słowy, otrzymał tylko zasady gry, a następnie grał przeciwko sobie wiele milionów razy (44 miliony gier w ciągu pierwszych dziewięciu godzin, według DeepMind)” [4]. Zamieszczony rysunek przedstawia wyniki szkolenia *Alpha Zero* (Rys. 1.). Można zauważyć,że początkowo program uczył się stosunkowo szybko, w mniej więcej trzysta tysięcznym kroku osiągnął poziom *Stockfisha,* natomiast później wolniej robił postępy.



Rysunek 1.Trening *AlphaZero* dla 700 000 kroków

. Źródło: www.researchgate.net

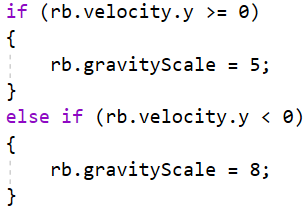
**2. Projekt i implementacja**

**2.1. Rozgrywka**

Gracz wciela się w postać robota. W zamyśle pracy miał on być humanoidalny. Celem tego zabiegu była próba utożsamienia się ze sterowaną postacią. Humanoidalne cechy głównej postaci stanowią kontrast do przeciwników przejawiających mniej lub bardziej inteligentne zachowanie, jednak ze względu na brak ludzkich kształtów ciężko utożsamiać ich z ludźmi.

Robot oczywiście porusza się, skacze, potrafi strzelać. Poruszanie się w lewo/prawo to po prostu zmiana komponentu transform po naciśnięciu odpowiedniego klawisza.

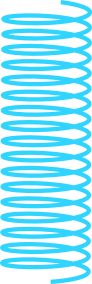
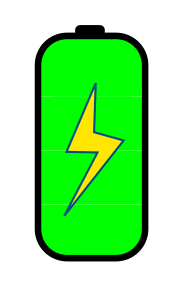
Skok natomiast wykonywany jest przy użyciu siły. Aby skakanie wyglądało jak najbardziej realistycznie w trakcie spadania w dół ustawiona została wyższa grawitacja tak aby robot szybciej spadał niż się wznosił (Listing 1). Udało się to poprzez sprawdzanie szybkości zmiany pozycji (*velocity*) robota. Gdy obiekt stoi na ziemi *velocity.y* = 0, gdy unosi się do góry *velocity.y*>0, gdy spada *velocity.y*<0.



Listing 1. Fragment kodu dotyczący zmiany grawitacji robota.

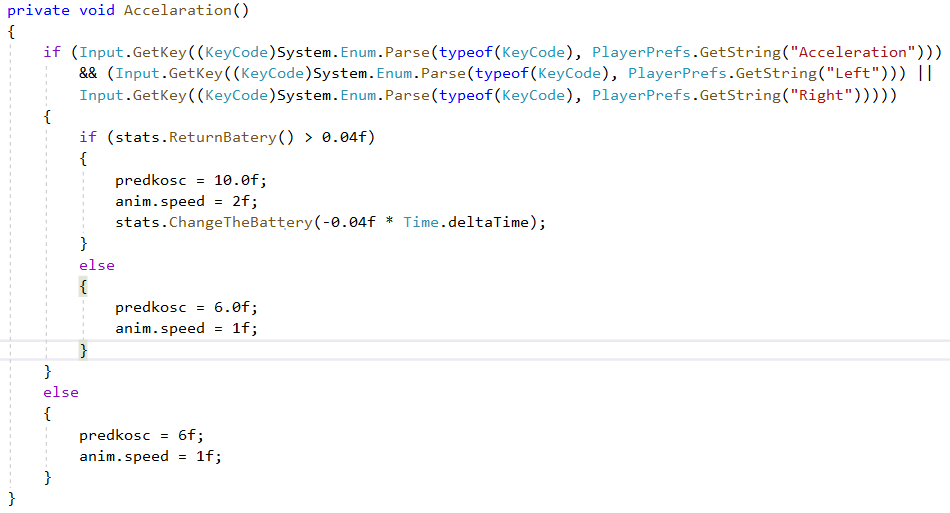
Samo poruszanie się, skok postaci jest zaimplementowane w funkcji *fixedUpdate*. Metoda ta jest używana do obliczeń fizycznych, ponieważ ma taką samą częstotliwość jak system fizyki (50 razy na sekundę). Jednak sprawdzanie, czy został naciśnięty odpowiedni klawisz odbywa się w *Update*. Metoda ta wywoływana jest co klatkę gry. Oznacza to, że zwykle będzie działać szybciej niż *fixedUpdate* [5]. Na podstawie własnych doświadczeń mogę stwierdzić ,że używanie *fixedUpdate* do sprawdzania danych wejściowych nie zawsze będzie działać. Postać może być obrócona w obie strony. Chcąc obrócić ja w tył wystarczyło zmienić współczynnik skali osi x na ujemny.

Gracz ma do dyspozycji dwa rodzaje ulepszeń. Jest to bateria oraz sprężynka.[Rys.2.]. Dzięki sprężynce można wyżej skakać natomiast zebranie baterii powoduje zwiększenie energii u robota. Energia wyczerpuje się podczas strzelania oraz zwiększania szybkości.



Rys 2. Ulepszenia w grze [Opracowanie własne]

Za mechanizm zwiększania szybkości odpowiada funkcja *Acceleration* (Listing 2). Sprawdza ona czy został naciśnięty klawisz przypisany do poruszania się w lewo/prawo oraz klawisz odpowiedzialny za przyspieszenie. Klawisze te zostały zapisane w klasie *PlayerPrefs*, która zostanie omówiona w następnym rozdziale. Następnie sprawdza czy poziom energii jest większy od 0.04 i jeśli tak prędkość zostaje zwiększona, a od stanu baterii odejmowana jest odpowiednia wartość. W celu stworzenia wrażenia, że robot rzeczywiście porusza się szybciej szybkość animacji również została zwiększona. Po wyczerpaniu się baterii lub puszczeniu odpowiedniego klawisza prędkość oraz animacja wracają do poprzednich wartości.



Listing 2. Fragment kodu odpowiedzialny za przyspieszenie [Opracowanie własne]

**2.2. Interfejs użytkownika**

Na interfejs składa się: menu główne, widok wyboru poziomu, panel menu, menu ustawień oraz panel zakończenia gry i zawarta w niej animacja.

Menu główne jest sceną rozpoczynającą grę, z której gracz może przejść dalej.

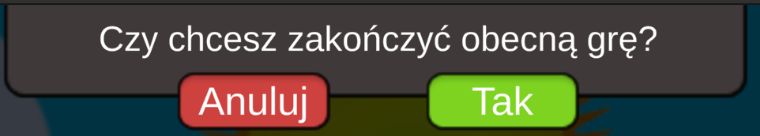
Widok wyboru poziomu umożliwia przejście do danego poziomu poprzez naciśnięcie w ten poziom.

Panel menu (Rys. 3.) jest prefabem tworzącym się po naciśnięciu przycisku menu. Jego naciśnięcie powoduje zatrzymanie gry poprzez ustawienie czasu gry na 0. Sam przycisk menu znika ze sceny i pojawia się dopiero po wznowieniu gry. Po naciśnięciu przycisku Wznów obiekt zostaje niszczony,a czas gry ponownie ustawiony na 1. Panel umożliwia powrót do menu głównego czy rozpoczęcie poziomu od początku.

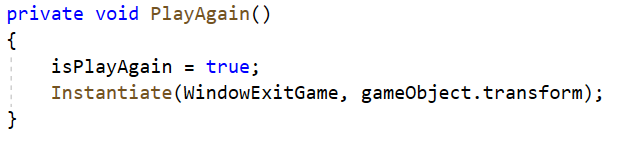
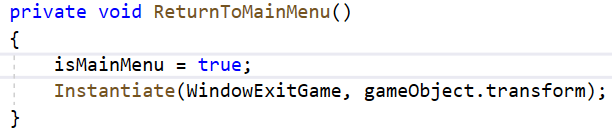


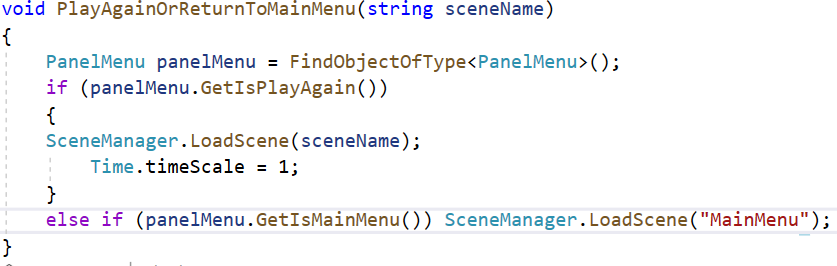
Rys. 3. Panel menu [opracowanie własne]

W celu zapobiegnięcia przypadkowego naciśnięcia któregoś z nich, przed wykonaniem danej czynności wyświetli się okno z pytaniem “Czy chcesz zakończyć obecną grę?” (Rys. 4.). Jest to obiekt, który po naciśnięciu *Tak* wykonuje dwie różne czynności. Może spowodować przejście do menu głównego lub rozpoczęcie poziomu od nowa. W tym momencie nasuwa się pytanie skąd program wie, którą czynność ma wykonać? Mianowicie po kliknięciu na odpowiedni przycisk zmienne bool *isPlayAgain*, *isMainMenu* zmieniają wartości na true, natomiast w klasie odpowiadającej za komunikat sprawdzane jest, która zmienna przyjmuje wartość true (Listing 3).



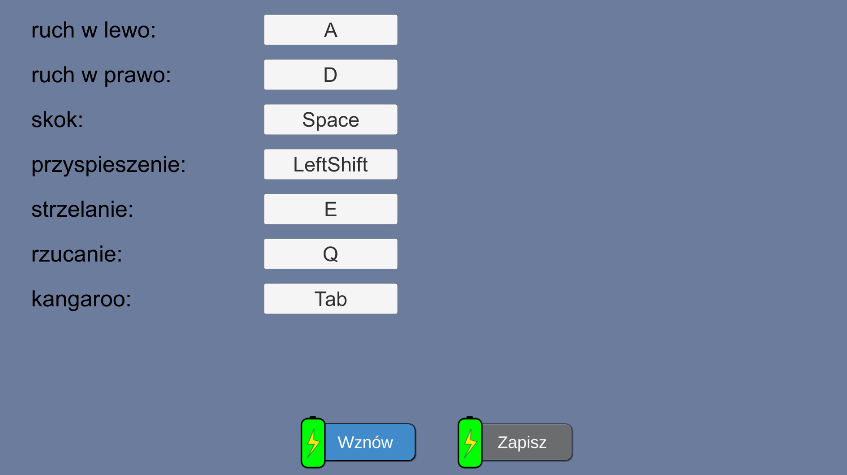
Rys. 4. Zabezpieczenie przed przypadkowym zamknięciem gry [opracowanie własne]



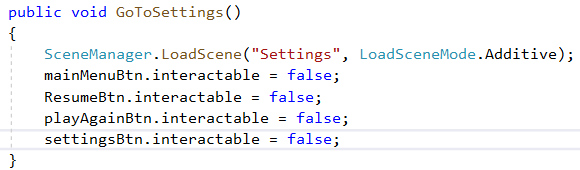


Listing 3. Fragment kodu odpowiedzialny za wyświetlanie okna [opracowanie własne]

W menu ustawień (Rys 5.) można dopasować sterowanie postacią według własnych preferencji. Można tam przejść z menu głównego lub z panelu menu. Zamieszczony rysunek przedstawia widok z panelu gry. Jeśli do ustawień przejdziemy z menu głównego ten widok poszerzy się o przycisk *Menu Główne*. Przycisk zapisz uaktywni się po tym gdy ustawienia zostaną zmienione. Opuszczenie gry i przejście do ustawień nie spowoduje utraty obecnego stanu gry. Dzieje się tak ponieważ, za pomocą *LoadSceneMode.Additive* do sceny gry zostaje dołączona scena ustawień bez zamykania obecnej sceny (Listing 4.). Aby uniknąć przypadkowego naciśnięcia, któregoś z przycisków panelu należało je zablokować.

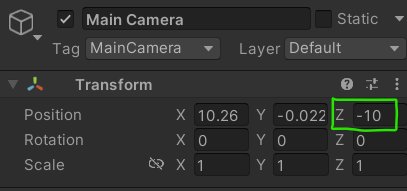
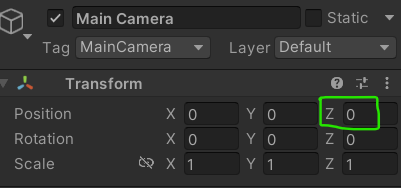


Rys. 5. Menu ustawień



Listing 4. Funkcja odpowiedzialna za przejście do ustawień [opracowanie własne]

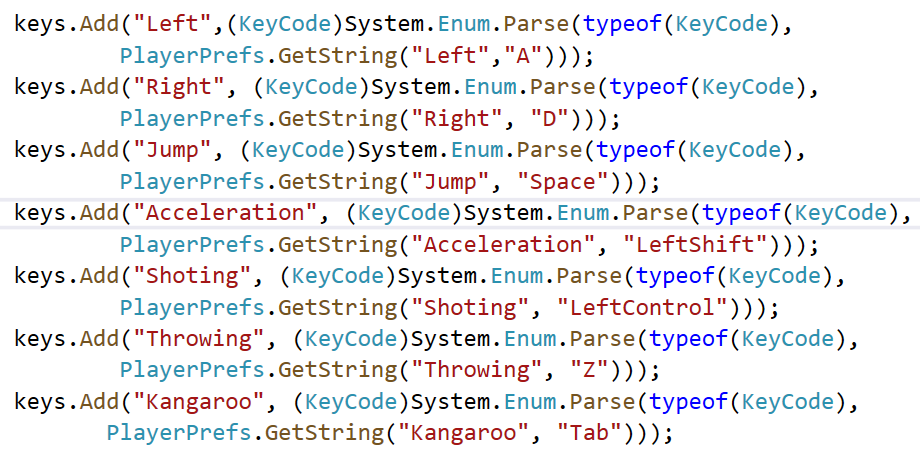
W tym momencie tak naprawdę widoczne są dwie sceny. Aby scena z grą pozostała niewidoczna ustawiłem jej kamerę dalej niż kamerę sceny ustawień. (Rys. 6.)



Rys. 6. Po lewo Pozycja kamery w scenie ustawień, po prawo Pozycja kamery w scenie gry

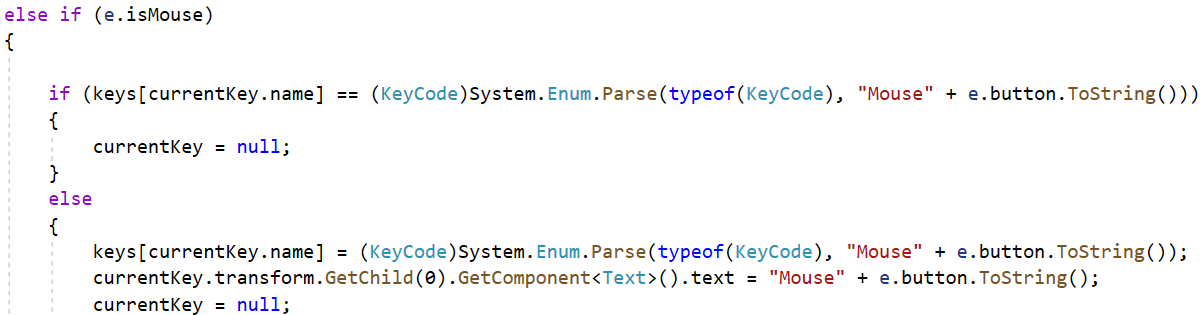
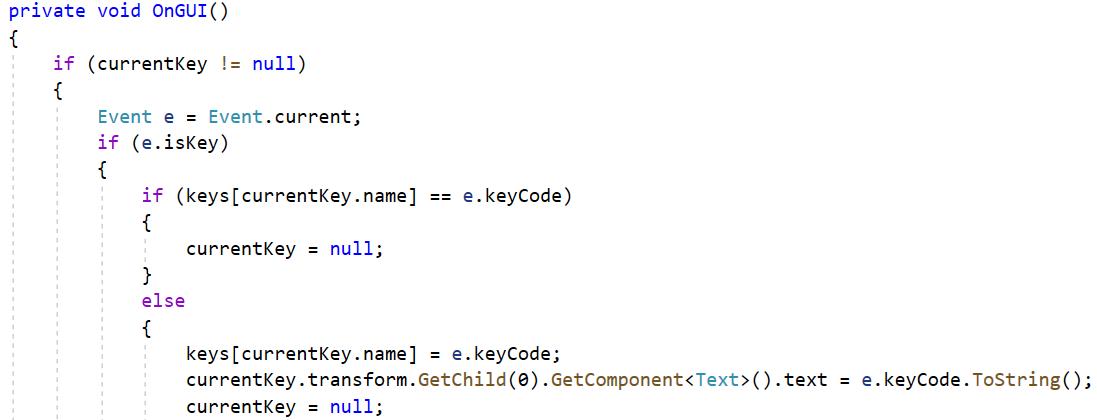
Gracz ma możliwość dostosowania klawiszy według własnych preferencji. Wystarczy nacisnąć przycisk i wybrać klawisz lub przycisk myszy. Wybrane przez nas ustawienia są dostępne także po ponownym uruchomieniu gry. Dzieje się tak, ponieważ dane są zapisywane w klasie *PlayerPrefs* . “ Klasa *PlayerPrefs* w Unity służy do zapisywania niewielkich ilości danych z gry na dysku, co pozwala na przechowywanie i przywoływanie pojedynczych ciągów, liczb całkowitych i zmiennych między sesjami rozgrywki.” [6] Aby umożliwić zapis tych danych utworzyłem słownik i dodałem do niego dane (Listing 5.).





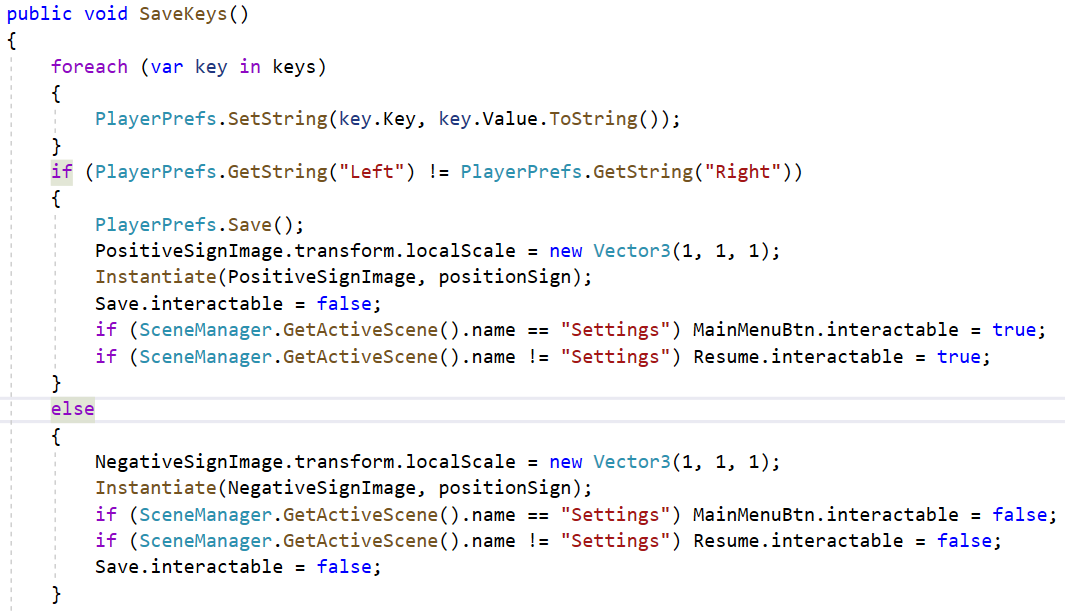
Listing 5. Utworzenie słownika i dodanie do niego danych [opracowanie własne]

W metodzie *OnGUI* (Listing 6.)sprawdzane jest czy został naciśnięty klawisz lub przycisk myszy.“*OnGUI* jest wywoływany do renderowania i obsługi zdarzeń GUI” [7](graficzny interfejs użytkownika). Metoda ta sprawdza czy naciśnięty klawisz jest inny niż dotychczas przypisany do konkretnej czynności, jeśli tak do słownika *keys* z kluczem obiektu *currentKey* zostaje przypisane wykonane zdarzenie, zaś w komponencie *Text* danego przycisku zmienia się napis. W przeciwnym razie do *currentKey* zostaje przypisana wartość *null*. Wartość w słowniku należało przekonwertować na typ *KeyCode*. W przypadku klawiszy myszy przed danym zdarzeniem należało dopisać *Mouse*, ponieważ te zdarzenia to wartości liczbowe, przykładowo prawy przycisk myszy to *2,* natomiast kod klucza *GetKey* interpretuje to jako *Mouse + number*.



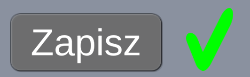
Listing 6.Metoda OnGui [Opracowanie własne(ale nie calkiem)]

W metodzie *SaveKeys* (Listing 7)do klasy *PlayerPrefs* zostają przypisane wartości ze słownika *keys* za pomocą pętli *foreach*. Następnie sprawdzane jest czy do klawiszy *Left* i *Right* nie został przypisany ten sam klawisz, jeśli przyciski są różne następuje zapis klasy *PlayerPrefs,* w przeciwnym razie zapis się nie dokonuje. W zależności od powodzenia zapisu tworzony jest obiekt “ptaszka” lub “x”. Ze względu na to że te obiekty są wykorzystywane w innej skali w panelu zakończenia gry, została ustawiona ich skala na 1 w każdym wymiarze. Po wykonanej akcji przycisk *Save* jest blokowany i uaktywnia się dopiero po zmianie dowolnego klawisza.



Listing 7. Funkcja SaveKeys [opracowanie własne]

Możliwość mapowania klawiszy została stworzona z myślą o wygodzie gracza. Jeśli z jakiegoś powodu życzy on sobie skakać, strzelać i poruszać się w prawo jednym klawiszem to ma taką możliwość. Natomiast nie ma możliwości poruszania się w prawo i w lewo jednocześnie. Z tego powodu zaimplementowałem warunek sprawdzający czy do tych tych funkcji nie został przypisany ten sam przycisk. Jeśli nie, nastąpi zapis danych, a obok przycisku widoczna będzi animacja rysowania *ptaszka* sugerująca, że zmiany zostały zapisane (Rys. 7.). W przeciwnym razie dane nie zostaną zapisane, wyświetli się animacja “X” (Rys. 7.), a przyciski powrotu do menu głównego i wznawiania gry zostaną zablokowane.



Rys. 7. Animacja

**2.3. Animacja**

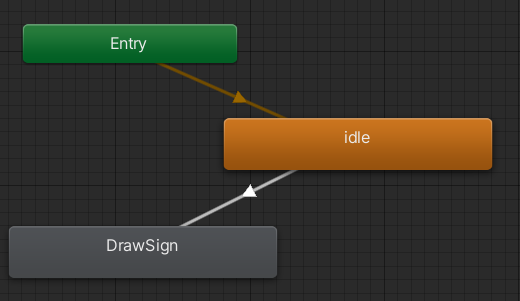
Chcąc odwzorować ruch obiektu należy go animować. Animacja jest sposobem na ożywienie postaci bądź obiektu. Animacja ma za zadanie stworzyć u odbiorcy złudzenie, że dana postać bądź obiekt rzeczywiście się porusza. W tym projekcie animacje to tak naprawdę sekwencje wyświetlanych po sobie kolejnych obrazków. W projekcie zawarta jest m.in. animacja rysowania znaku ptaszka. (Rys. 8.)



Rys. 8. Znak ptaszka wykonany w programie inkscape

Przebiegła ona w następujący sposób:

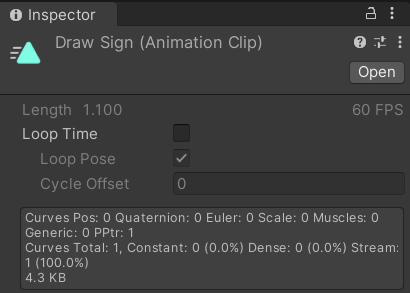
1. Przygotowanie w programie Inkscape symbolu,
2. W programie Gimp usuwanie kolejnych części znaku, aż do niemalże całkowitego usunięcia obrazu
3. Stworzenie animacji DrawSign.anim i co 0.01 sekundy dodanie kolejnych obrazków w kolejności od najmniej wypełnionego obrazu
4. Za pomocą kontrolera animacji stworzenie maszyny stanów animacji (Rys. 9.).



Rys. 9. Maszyna stanów animacji

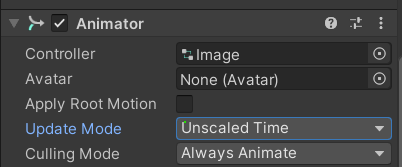
W ustawieniach animacji,domyślnie jest ona zapętlona. W tym przypadku nie było to moim

celem, dlatego też należało odznaczyć opcje *loopTime*. (Rys. 10.)



Rys. 10. Ustawienia animacji [opracowanie własne??]

Animacja ta jest wywoływana w momencie gdy skala, w której upływa czas ustawiona jest na 0, z tego powodu musiałem wybrać opcję *UnscaledTime* w *UpdateMode* w komponencie *Animator* dołączonych do obiektu. (Rys. 11.)



Rys. 11. Komponent *Animator* obiektu [opracowanie własne??]

**2.4. Wykorzystane technologie**

**2.4.1. Unity**

“Jest to silnik do tworzenia trójwymiarowych oraz dwuwymiarowych gier komputerowych lub innych materiałów interaktywnych, takich jak wizualizacje czy animacje” [10]. Unity posiada szeroki zakres obsługiwanych platform, od internetu i urządzeń mobilnych po komputery PC i konsole. Posiada jedną z największych i aktywnych społeczności i jest stosunkowo łatwy do nauki. Silnik ma wiele narzędzi ułatwiających ponowne wykorzystanie zasobów i kodu takich jak: Asset Store oraz Package Manager. Z w.w. powodów zdecydowałem się użyć właśnie tego silnika.

**2.4.2. Visual Studio**

Jest to “zintegrowane środowisko programistyczne (IDE) firmy MIcrosoft”[10]. Visual Studio jest zintegrowane z silnikiem unity i służyło mi do pisania skryptów w języku c#.

**2.4.3. Git/TortoiseGit/GitHub**

Git to najpopularniejszy system kontroli wersji na świecie. System kontroli wersji umożliwia zapis kolejnych wersji projektu. Niewątpliwą zaletą gita jest bezpieczeństwo. Jest dobrze zorganizowany przez co ciężko jest stracić wprowadzone zmiany, w łatwy sposób można znaleźć poprzednie wersje systemu. Umożliwia rozgałęzienie projektu a następnie scalenie gałęzi. Ta opcja doskonale się sprawdziła gdy na jednej gałęzi pracowałem nad poziomami, przeciwnikami itd., a na drugiej odbywało się uczenie przez wzmacnianie. TortoiseGit to właściwie graficzny interfejs z opcjami gita. Po prostu ułatwia pracę z gitem. Według wikipedii Github to hostingowy serwis internetowy przeznaczony do projektów programistycznych wykorzystujący system kontroli wersji Git. Pozwolę się nie zgodzić z jednym słowem: programistycznych. Git równie dobrze może służyć do zwykłych plików tekstowych, takich jak ta praca dyplomowa, a prawdopodobnie też do każdego innego rodzaju pliku. GitHub umożliwia po zatwierdzeniu swojej pracy lokalnie, zsynchronizowanie kopii repozytorium z kopią na serwerze. Mówiąc inaczej pracę wykonaną na swoim komputerze, można przenieść na serwer zdalny przez co np., po utracie danych z komputera nie stracimy swojej dotychczasowej pracy.

**3. Zastosowanie sztucznej inteligencji**

W celu osiągnięcia inteligentnego zachowania przeciwników zaimplementowany został w tej pracy algorytm *pathfinding*, maszyna stanów skończonych (Finite State Machine), uczenie przez wzmacnianie (Reinforcement learning, RL) oraz pole widzenia.

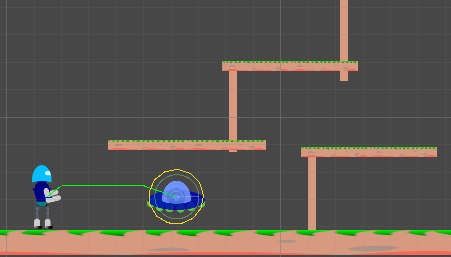
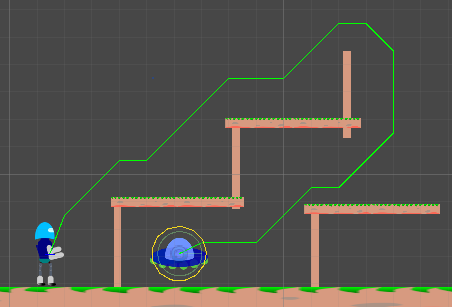
**3.1. Pathfinding**

Algorytm A\* to jeden z najpopularniejszych algorytmów *pathfindingu*, który wykorzystuje heurystykę do szacowania kosztu podróży między wierzchołkami grafu. Ten algorytm jest często stosowany w grach, w których trzeba znaleźć najkrótszą ścieżkę między dwoma punktami**.**

Algorytm A\* działa w następujący sposób:

1. Inicjujemy zbiór otwartych węzłów, których koszt zostanie obliczony, oraz zbiór zamkniętych węzłów, które już odwiedziliśmy.
2. Dodajemy do zbioru otwartych węzłów węzeł startowy i ustawiamy jego koszt na zero.
3. Wybieramy węzeł z otwartego zbioru, który na najniższy koszt f(n) = g(n)+h(n), gdzie g(n) to koszt dotarcia do danego węzła z węzła startowego, h(n) to szacowany koszt dotarcia z danego węzła końcowego, a f(n) to łączny koszt dotarcia do danego węzła.
4. Sprawdzamy, czy wybrany węzeł jest węzłem końcowym. Jeśli tak, kończymy algorytm i zwracamy ścieżkę.
5. W przeciwnym razie dodajemy wybrany węzeł do zbioru zamkniętych węzłów i rozważamy wszystkie jego sąsiednie węzły.
6. Dla każdego sąsiedniego węzła obliczamy koszt g(n) i h(n), a następnie obliczamy łączny koszt f(n).
7. Jeśli sąsiedni węzeł nie należy do zbioru ani otwartych ani zamkniętych węzłów, dodajemy go do zbioru otwartych węzłów i ustawiamy jako węzeł rodzica dla tego węzła.
8. Jeśli sąsiedni węzeł należy już do zbioru otwartych węzłów, aktualizujemy jego koszt g(n) oraz ustawiamy go jako węzeł rodzica dla tego węzła, jeśli nowy koszt jest mniejszy niż poprzedni.
9. Powtarzamy kroki 3-8, aż do momentu, gdy węzeł końcowy zostanie dodany do zbioru zamkniętych węzłów. wtedy kończymy algorytm i zwracamy znalezioną ścieżkę.

W celu zaimplementowania algorytmu A\* wykorzystany został pakiet *pathfinding* dla *Unity*. W grze został stworzony statek UFO (rys). Algorytm A\* posłużył tutaj do znalezienia najkrótszej drogi do celu. Aby zaimplementować ten algorytm należało podać cel do którego ma dążyć obiekt UFO, w tym przypadku jest to gracz, podać warstwę bądź warstwy, które musi omijać oraz zdefiniować średnicę przez jaką może się przelecieć.

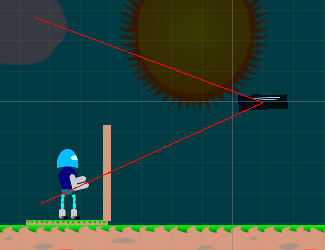
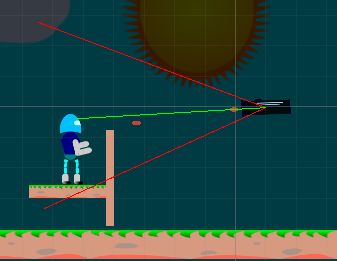


Rys. Widok sceny ze statkiem UFO. Zielona linia pokazuje trasę jaką przebędzie statek

**3.1. Pole widzenia**

Ponieważ jest to element, który wiążę się z następnym rozdziałem został omówiony tutaj. Pole widzenia w grze jest niczym innym jak obszarem, który widzi dany obiekt. Jeśli w tym obszarze znajdzie się inny obiekt, przykładowo gracz, to nasz przeciwnik podejmuje odpowiednie działania. (Rys. 21.)

Jak działa pole widzenia? Ustawiamy promień i kąt widzenia oraz warstwy: jedna z nich określa gracza, druga określa przeszkody. Od obiektu z polem widzenia rzucany jest promień, jeśli natrafi on bez żadnych przeszkód, na naszą postać, a raczej na warstwę na której jest, jesteśmy w polu widzenia. Natomiast gdy promień natrafi najpierw na przeszkodę jesteśmy poza polem widzenia.

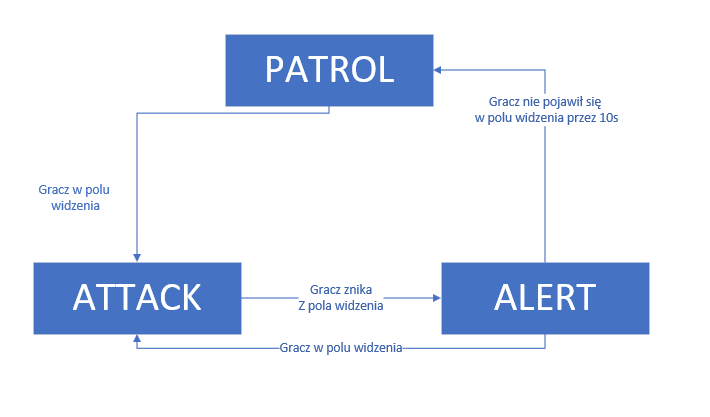


Rys. 12. Po lewo gracz w polu widzenia, po prawo poza polem widzenia [opracowanie własne]

**3.2. Maszyna stanów skończonych**

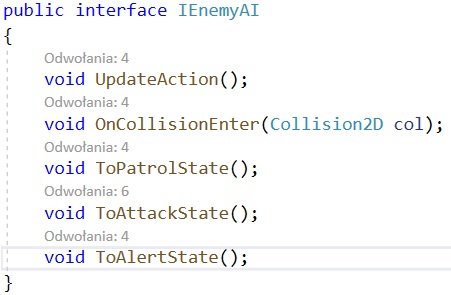
“Najpowszechniej i najszerzej stosowana metoda realizacji sztucznej inteligencji w grach komputerowych to Maszyny Stanów Skończonych (ang. Finite-State Machine – FSM). W polskim tłumaczeniu pojawia się też nazwa: automat o skończonej liczbie stanów lub w skrócie automat skończony.”[8]

Zasada działania jest dość prosta. Mamy określoną skończoną liczbę stanów, pomiędzy którymi są przejścia, realizowane po wykonaniu pewnych warunków (Rys. 23.).



Rys. 23 Graficzne przedstawienia maszyny stanów i warunków przejść [Opracowanie własne]

W celu stworzenia maszyny stanów został utworzony interfejs, w którym zaimplementowane zostały wszystkie potrzebne metody (Listing 8.).



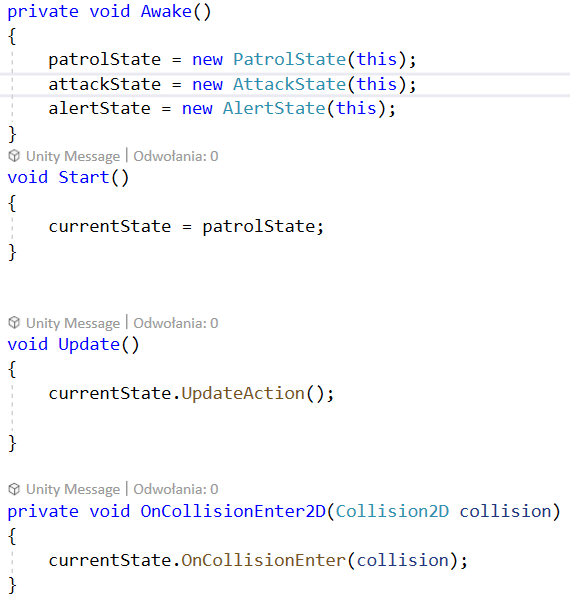
Listing 8. Interfejs z niezbędnymi metodami [opracowanie własne na podstawie [**numberYT**]]

Następnie trzy klasy dziedziczące po IEnemyAI , w których zaimplementowałem ww. metody (listing 9.). Każda z nich odpowiada za konkretny stan. Metody zaczynające się od “To” odpowiadają za przejścia do konkretnych stanów. *UpdateAction* odpowiada za to co będzie się dziać w każdej klatce podczas działania gry. Przykładowo dla klasy *PatrolState* metoda ta odpowiedzialna jest za patrolowanie terenu, a dokładniej za obracanie obiektu wokół własnej osi. Kiedy gracz znajdzie się w polu widzenia następuję przejście do stanu atakowania. *OnCollisionEnter(Collision2D)* odpowiada za to co stanie się jeśli coś wejdzie w kolizje z obiektem. Dla klasy *PatrolState* będzie to przejście do ataku po zderzeniu się z graczem lub wystrzelonym przez niego pociskiem.



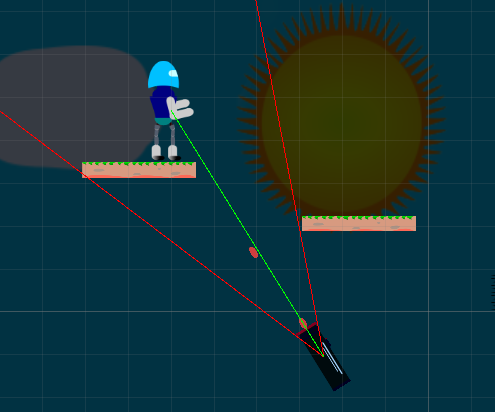
Listing 9. Jedna ze wspomnianych klas w tym przypadku klasa *PartolState* [opracowanie własne]

Na koniec stworzyłem skrypt, (Listing 10.) który będzie tym wszystkim zarządzał i przypisałem go do obiektu przeciwnika. W metodzie *Update* wywoływany jest aktualny stan.



Listing 10. Fragment kodu odpowiedzialny za aktualnie wywoływany stan [opracowanie własne]

Rezultat:



Rys. 13. Autogun w stanie atakowania

Zaletą stosowania maszyny stanów skończonych jest łatwa możliwość rozbudowy maszyny o kolejne stany oraz dodawanie kolejnych warunków.

**3.3. Uczenie ze wzmocnieniem**

Zadaniem uczenia przez wzmacnianie jest trenowanie modelu, tak aby dojść do optymalnego rozwiązania problemu poprzez samodzielne podejmowanie decyzji.

Składa się z :

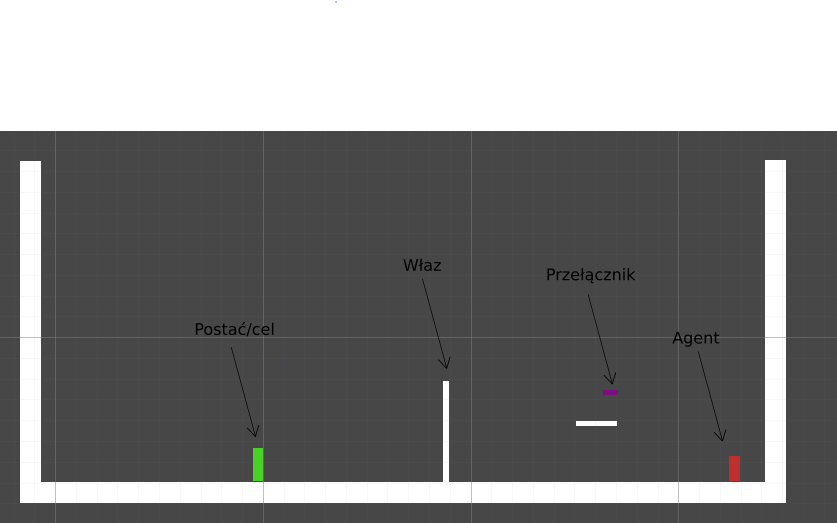
- Środowiska , z którym agent będzie wchodzić w interakcje, aby nauczyć się osiągać cel lub wykonywać działania,

- Nagrody dodatniej, jeśli podejmowane decyzje przez „agenta” są pożądane

- Nagrody ujemnej w przeciwnym razie.

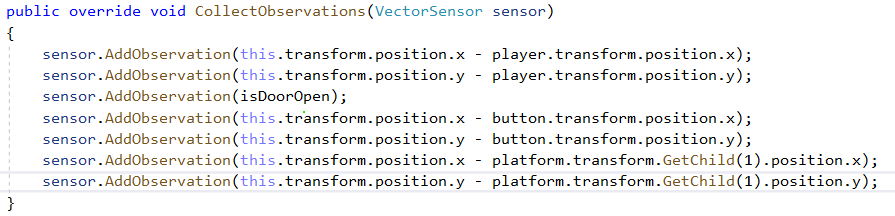
W celu wyszkolenia agenta użyte zostało Unity ML-agent. “Zestaw narzędzi Unity Machine Learning Agents Toolkit (ML-Agents) jest oprogramowaniem typu open source projekt, dzięki któremu gry i symulacje mogą służyć jako środowiska dla szkolenie inteligentnych agentów. Dostarcza implementację (oparte na PyTorch) najnowocześniejszych algorytmów, aby umożliwić twórcom gier łatwe szkolenie inteligentnych agentów do gier.”[9]

Na początek warto zacząć od tego na czym polega problem. Zadaniem tak zwanego agenta jest wyeliminować postać z gry poprzez zderzenie się z nią. Na jego przeszkodzie będzie stał właz, który można podnieść poprzez przełączenie przycisku. Przed przystąpieniem do szkolenia dobrze jest zduplikować środowisko szkolenia przykładowo kilkadziesiąt razy. Ma to na celu zwiększenie liczby wykonywanych kroków co przekłada się na zmniejszenie czasu szkolenia.



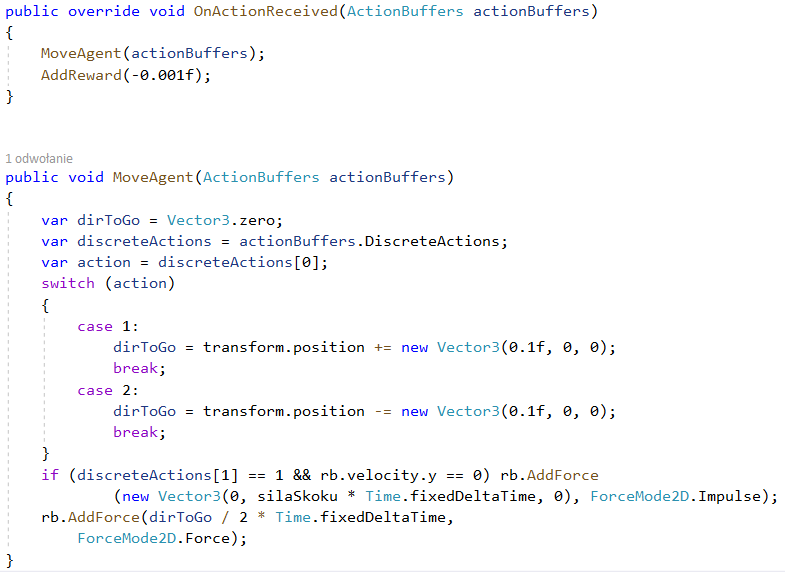
Rys 14. Środowisko szkolenia agenta. Kształty zostały zmienione. [opracowanie własne]

Program posiada 7 obserwacji. Obserwuje odległości x i y pomiędzy agentem, a poszczególnymi elementami środowiska oraz to czy można przejść pod włazem poprzez zmienną typu *bool isDoorOpen* (Listing 11.). Na podstawie tych obserwacji agent będzie podejmował decyzje. Kolejność tych obserwacji jest ważna i nie można jej zmieniać ani w trakcie uczenia ani tym bardziej po nauczeniu.



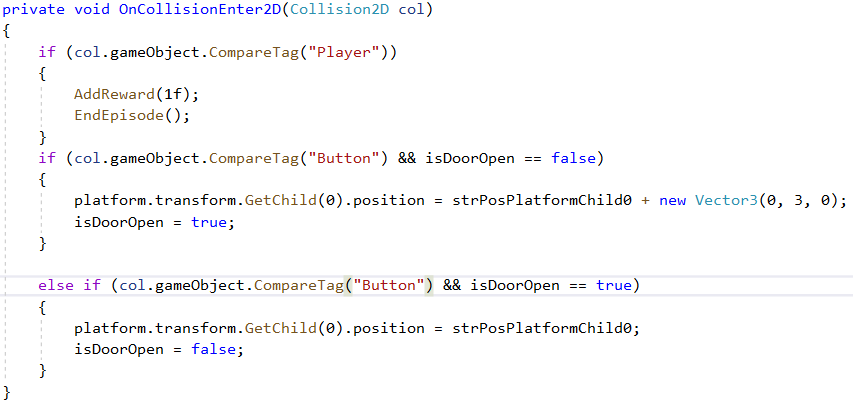
Listing 11. Fragment kodu przedstawiający implementacje obserwacji [opracowanie własne]

Funkcja *OnActivationReceived* (Listing 12.) służy do implementowania działań, które może wykonać Agent. W funkcji tej zaimplementowałem trzy akcje: *case 1* - ruch w prawo, *cace 2* - ruch w lewo oraz skok. Podczas wykonywania ruchu w lewo czy prawo komponent *transform.position* jest zmieniany o 0. Zwiększanie tej wartości powodowało już pewne problemy Przed wykonaniem skoku sprawdzana jest prędkość *velocity.y* tak aby uniemożliwić agentowi skakanie z powietrza. W funkcji tej odejmuję od nagrody 0.001 punkta. Celem tego zabiegu jest zmuszanie agenta do jak najszybszego osiągnięcia celu.



Listing 12. Fragment kodu odpowiedzialny za działanie agenta [opracowanie własne]

Nagrody przydziela się za pomocą funkcji *AddReward* lub *SetReward*. Różnią się one tym, że pierwsza dodaje bądź odejmuje punkty z ogólnej puli w danym kroku (epizodzie), druga zaś ustawia liczbę punktów na konkretną wartość. Po kolizji agenta i obiektu o tagu *Player* do nagrody dodawany jest 1 punkt, po czym następuję przejści do kolejnego kroku. W funkcji odpowiedzialnej za działania podejmowane po kolizji z agentem zaimplementowany został mechanizm podnoszenia oraz opuszczania włazu. Metoda ta sprawdza czy agent wszedł w kolizje z obiektem o o tagu *Button* oraz zmienną *isDoorOpen* i w zależności od niej podnosi lub opuszcza właz.



Listing 13. Fragment kodu odpowiedzialny za działania podejmowane po zderzeniu się agenta z innymi obiektami [opracowanie własne]

*EndEpisode* jak sama nazwa wskazuje kończy episod. Kończy się on po upływie określonego czasu gdy agent osiągnie liczbę *MaxSteps* lub prościej mówiąc gdy zadecyduje o tym programista. Po czym wykonuje funkcje *OnEpisodeBegin*, w której to określa się ustawienia m.in. współrzędne danych elementów.

Proces szkolenia odbywa się już bez naszej ingerencji. Po prostu w konsoli należy uruchomić szkolenie i włączyć grę. W konsoli można zauważyć dwie wartości: *meanReward* (średnia nagród) oraz *stdReward* (odchylenie standardowe). Im większa wartość *meanReward* tym lepiej, z oczywistych powodów. *StdReward* natomiast powinno być jak najniższe, ponieważ oznacza to, że agent jest powtarzalny w swoich działaniach. Szkolenie można zakończyć w dowolnym momencie naciskając ctrl+c. Wtedy otrzymujemy gotowy do użycia model agenta. Ze względu na mało wygodną formę przedstawianych danych w konsoli, zostały on zaprezentowane w postaci tabeli.

| Liczba kroków | Czas w sekundach | Średnia nagroda | Odchylenie standardowe nagrody |
| --- | --- | --- | --- |
| 100000 | 102.327 | -4.532 | 1.601 |
| 300000 | 283.109 | -2.145 | 2.627 |
| 350000 | 327.074 | -0.657 | 2.250 |
| 400000 | 372.545 | 0.262 | 1.516 |
| 500000 | 462.518 | 0.751 | 0.541 |
| 2000000 | 1977.469 | 0.858 | 0.254 |
| 7500000 | 7 556.57 | 0.865 | 0.327 |
| 10350000 | 9 852,489 | 0.877 | 0.208 |
| 12900000 | 11 910,349 | 0.886 | 0.118 |
| 12950000 | 11 951,189 | 0.882 | 0.119 |

Rys. 25. Tabela przedstawiająca proces szkolenia

W tabeli zostały zamieszczone tylko niektóre dane, obrazujące mniej więcej cały proces szkolenia. Można z nich wywnioskować, że agent stosunkowo szybko nauczył się czegokolwiek, już po 373 sekundach średnia nagroda była dodatnia, a w pięćset tysięcznym kroku zaczęła przypominać nagrody z ostatnich wierszy tabeli. Odchylenie standardowe wartości 1.5-0.5 było jednak dalekie od zamierzonego, wartości te oznaczały, że agent jest niestabilny. Dalsze doskonalenie agenta było już dość długie. Ponieważ po dziesięciomilionowym kroku wartości były zadowalające i przede wszystkim powtarzalne, proces szkolenia został zakończony.

**Podsumowanie**

Przede wszystkim po wykonaniu oraz napisaniu pracy stwierdzam iż należało zacząć pracę wcześniej. Nawet z pozoru łatwe zadania okazywały się pracochłonne lub też musiałem zrobić coś czego wcześniej nie przewidziałem. Zdecydowanie pracę ułatwiłoby mi też wcześniejsze jej rozplanowanie. Oczywiście niemożliwe było całkowite rozplanowanie funkcjonowania gry, poszczególne elementy gry były często wymyślone spontanicznie. Ustalenie na samym początku planu gry, pozwoliłoby uniknąć robienia czegoś bez potrzeby. W swoim projekcie zdecydowałem się użyć omówionego wcześniej systemu kontroli wersji-Git. Narzędzie to umożliwiło mi wygodną pracę nad projektem bez martwienia się o utratę go. Nie wyobrażam sobie pracy nad jakimkolwiek projektem bez tego typu narzędzia. W swojej pracy poruszyłem temat sztucznej inteligencji. Moim zdaniem jej udział w branży gier wideo będzie coraz większy. Omówione w pracy uczenie maszynowe zwiększa wydajność. Dzięki zastosowaniu uczenia maszynowego i głębokiego uczenia, można stworzyć przeciwników, którzy będą zachowywać się bardziej realistycznie i przewidywalnie. W ten sposób gra staje się bardziej wymagająca i interesująca dla graczy. Sztuczna inteligencja dobrze się sprawdza do tworzenia tekstów dialogów, misji czy samych pomysłów jak stworzyć grę, czego przykładem jest Chat GPT. Mimo to uważam, że sztuczna inteligencja nie zastąpi całkowicie tradycyjnych technik programowania, przynajmniej nie w najbliższej przyszłości. Jednym z argumentów to potwierdzających jest fakt iż wytrenowanie modelu w *RobotGame* było dość długie. Uczenie przez wzmocnienie może być dobrym sposobem na udoskonalenie konkretnej rzeczy.

Bibliografia:

//[1] Moja historia gier platformowych - część 1 - GeneticsD

[2] Uczenie maszynowe – Wikipedia, wolna encyklopedia

[3] 20 lat po DeepBlue: Jak sztuczna inteligencja rozwinęła się od czasu podboju szachów -

- Scientific American. Larry Greenmeier 2 czerwca 2017

[4] AlphaZero - Silniki szachowe - Chess.com https://www.chess.com/terms/alphazero-chess-engine?page=3

[5] Carrillo U.,Update vs. FixedUpdate vs. LateUpdate in Unity - [Update vs. FixedUpdate vs. LateUpdate in Unity - LogRocket Blog](https://blog.logrocket.com/update-vs-fixedupdate-vs-lateupdate-in-unity/) [12.10.2022]

[6] Jak korzystać z Player Prefs w Unity - Game Dev Beginner: John French, 1 listopad 2021

[7] Unity - Skrypty API: MonoBehaviour.OnGUI() (unity3d.com)

[8] Jak powstaje sztuczna inteligencja w grach komputerowych - Just Geek IT (justjoin.it)

8 października 2018 [Marek Winiarski](https://geek.justjoin.it/author/marek-winiarski/)

[9] https://github.com/Unity-Technologies/ml-agents

[10] Unity (silnik gry) – Wikipedia, wolna encyklopedia

[11] Microsoft Visual Studio – Wikipedia, wolna encyklopedia

[numberUM] Matt R. Cole, Uczenie maszynowe w C#. Szybkie, sprytne i solidne aplikacje. wydawnictwo: Helion [06.11.2019]

[numberSN] Sutton S.R. i Barto G. A. Reinforcement Learning: An Introduction"[] wydawnictwo: The MIT Press. 2018 rok.

[numberUPW] Bajaj P. <https://www.geeksforgeeks.org/what-is-reinforcement-learning/> [07.03.2023]

[numberYT] Fabryka twórców gier, <https://www.youtube.com/watch?v=xtQVqqmwGac> [16.11.2016]