Wydział
Informatyki

Tomasz Kerber

numer albumu: 41395

kierunek studiów: Informatyka

specjalność: Systemy komputerowe i oprogramowanie

forma studiów: studia stacjonarne

PROJEKT I IMPLEMENTACJA WYBRANEJ GRY ZRĘCZNOŚCIOWEJ, UMOŻLIWIAJĄCEJ ZASTOSOWANIE UCZENIA ZE WZMOCNIENIEM DO POSZUKIWANIA STRATEGII GRACZA

DESIGN AND IMPLEMENTATION OF A SELECTED ARCADE GAME THAT ALLOWS APPLICATION OF THE REINFORCEMENT LEARNING TO SEARCH FOR A PLAYER'S POLICY

praca dyplomowa inżynierska

napisana pod kierunkiem:

dr. hab. inż. Marcina Plucińskiego, prof. ZUT

Katedra Sztucznej Inteligencji i Matematyki Stosowanej

Data wydania tematu pracy:

Data dopuszczenia pracy do egzaminu: .....

(uzupełnia pisemnie Dziekanat)

Szczecin, 2022

Oświadczenie
autora pracy dyplomowej

Oświadczam, że praca dyplomowa inżynierska pn. *Projekt i implementacja wybranej gry zręcznościowej, umożliwiającej zastosowanie uczenia ze wzmocnieniem do poszukiwania strategii gracza* napisana pod kierunkiem dr. hab. inż. Marcina Plucińskiego, prof. ZUT jest w całości moim samodzielnym autorskim opracowaniem sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych. Złożona w dziekanacie Wydziału Informatyki treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej.

Oświadczam ponadto, że złożona w dziekanacie praca dyplomowa ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczelniach wyższych.

Podpis autora: ..Tomasz Kołtun.....

Szczecin, dnia: ..10.06.2022 r.....

Streszczenie

Celem tej pracy jest opracowanie prostej gry zręcznościowej, w której będzie można zademonstrować uczenie ze wzmocnieniem do poszukiwania strategii gracza. Uczenie ze wzmocnieniem to dziedzina sztucznej inteligencji, w której uczącemu się agentowi (uczonemu systemowi) nie dajemy żadnych gotowych danych do uczenia. Wszystkie informacje, będzie musiał sam zdobyć, eksplorując środowisko. Dlatego postanowiłem napisać prostą grę w języku C++, a następnie wyniki wybranego uczenia ze wzmocnieniem i grę zaprezentować.

Słowa kluczowe: Uczenie ze wzmocnieniem , komputerowe gry zręcznościowe

Abstract

The goal of this work is to develop a simple arcade game in which we will be able to demonstrate reinforcement learning for player strategy search. Reinforcement learning is a branch of artificial intelligence in which the learning agent(learned system) is not given any ready-made data to learn. All the information it will have to acquire on its own by exploring the world. Therefore, I decided to write a simple game in C++ language and then the results of the selected reinforcement learning and the game will be presented.

Keywords: teaching with reinforcement , computer based arcade games

Spis treści

Spis rysunków

Wstęp

W czasach, w których żyjemy praktycznie nie ma osoby, która nie miała styczności z grami komputerowymi. Szacuje się, że na Ziemi w gry gra około 2,5 miliarda osób. Są one częścią życia wielu z nas. Z tego powodu, rynek właśnie tej oto rozrywki jest ogromny. Każdego roku powstaje wiele produkcji, więc każdy pasjonat gier znajdzie coś dla siebie. Niektóre gry są na tyle popularne, że rozwinął się w nich e-sport, w którym gracze rywalizują indywidualnie lub grupowo o tytuł najlepszych w danej grze i nawet o prawdziwe nagrody pieniężne [9].

W każdej grze gracze muszą rozgryźć jakąś zagadkę albo pokonać drugą osobę lub bota. Bot to program komputerowy, który jest zaprogramowany do rozwiązywania pewnych zagadnień na potrzeby człowieka. W grach jest to przeciwnik, który zastępuje prawdziwego człowieka. Nasz bot może też grać sam w gry komputerowe szukając optymalnych strategii do przejścia gry. Programowaniem takiego zachowania, symulującego inteligentne podejście do gry, zajmuje się jedna z gałęzi informatyki, którą jest sztuczna inteligencja.

Dziedzina ta posiada pełen wachlarz algorytmów, które potrafią ożywić bota i nadać mu cechy ludzkie. Trzeba pamiętać aby programując takie boty nie były one ani za silne ani za słabe. Jest to bardzo ważne, ponieważ gdy nasz sztuczny przeciwnik będzie rozważał zawsze najlepsze posunięcia, to może to zniechęcić gracza i może stracić on ochotę do gry. Taka sama sytuacja może się zdarzyć grając w grze ze zbyt słabym przeciwnikiem, ale tutaj niechęć użytkownika takiej gry może wynikać z braku satysfakcji pokonania bota, więc najlepiej byłoby gdyby algorytmy w nowo tworzonych grach nadawały przeciwnikom zachowania przypominające ludzkie.

Z pomocą przy tworzeniu takich sztucznych przeciwników przychodzą nam algorytmy sztucznej inteligencji zwane uczeniem ze wzmocnieniem. Jest to bardzo popularna gałąź informatyki ze względu na łatwość wdrożenia takich algorytmów do swoich prac i wyniki jakie daje.

Celem pracy jest implementacja prostej gry zręcznościowej, w której będzie możliwość zastosowania wybranego przeze mnie algorytmu uczenia przez wzmocnienie, którym nauczymy agenta potrzebnej strategii, aby wygrać grę. Następnie zostaną przeprowadzone badania, dzięki którym zostaną wybrane najlepsze parametry algorytmu i zostanie zaprezentowana skuteczność otrzymanego bota.

1. Czym jest gra zręcznościowa i na czym polega występująca w niej strategia gracza?

1.1 O początkach gier

W dzisiejszym świecie chyba nikt nie jest w stanie wyobrazić sobie codzienności bez komputerów. Towarzyszą nam w każdym aspekcie życia, od smartfonów, przez medycynę, po wyprawy kosmiczne. Przez lata ich funkcje oraz towarzyszące temu oprogramowania przeżywają rozwój. Od etapu wykonywania skomplikowanych prac, czy obliczeń, po sprawianie przyjemności człowiekowi. W ten sposób pojawiły się gry komputerowe, które z prostych algorytmów ewoluowały do skomplikowanych programów z wysoce rozwiniętą grafiką. Wśród nich wyodrębniamy gry logiczne, edukacyjne, symulacyjne, sportowe, strategiczne, fabularne, przygodowe oraz zręcznościowe. To właśnie te ostatnie stanowią przedmiot niniejszej pracy inżynierskiej.

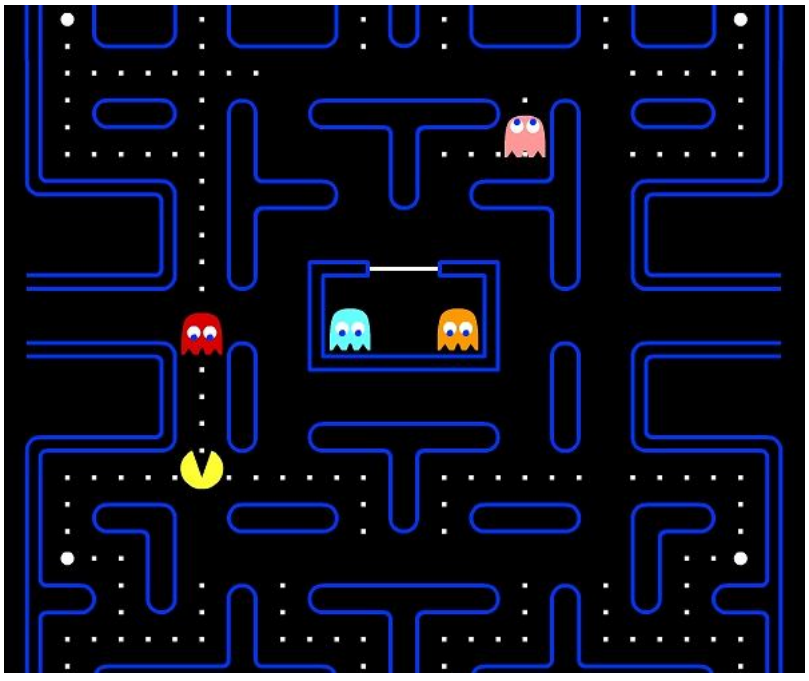
Nie sposób jednak nie wspomnieć o początkach gier komputerowych. Mają one miejsce już w 1942 roku. Program Analogowy symulator pocisku rakietowego stworzony przez dwóch Amerykanów Thomasa A. Goldsmitha Jr. i Estle Raya dał początek przemysłowi gier. Jednak odsłona gier w wydaniu graficznym nastąpiła dopiero po wynalezieniu komputera EDSAC. W dalszym etapie pojawiła się pierwsza gra wykorzystująca kontrolery do sterowania, co miało miejsce w roku 1958.

Wśród gier tego typu wyróżniamy te o grafice dwu- oraz trzywymiarowej. Grafika 2D swój początek ma w latach 50-tych 20-tego wieku. W jej przypadku nie ma możliwości tworzenia kształtów i efektów charakterystycznych dla przestrzeni trójwymiarowych (oświetlenie, cienie). Kreowana jest za pomocą warstw lub ich odległości od obserwatora. Grafika oparta na silniku 3D jest bardziej atrakcyjna dla gracza, jednak rendering jest procesem wysoce skomplikowanym i czasochłonnym, stąd większość gier obecnie produkowanych opiera się na bibliotekach np. DirectX. Liczne efekty, jak światło czy cieniowanie wpływa na realizm grafiki. W tej pracy inżynierskiej zostanie stworzona gra oparta na grafice dwuwymiarowej.[3]

Rysunek 1.1: Automaty do gier popularne w latach 80.

1.2 Co to jest gra zręcznościowa?

Gry zręcznościowe zwane są popularnie zręcznościówkami. Ich zwycięskie ukończenie zależne jest od sprawności gracza w operowaniu kontrolerem, czyli urządzeniem wejściowym służącym do sprawowania kontroli nad przebiegiem gry (klawiatura, dżojstic, mysz, gamepad). W grach tego typu liczy się refleks gracza, ciągłość akcji. Pierwszą grą z tego gatunku był PONG z 1972 roku. Ten rodzaj gier szczególny rozwój i popularność zdobyły w latach osiemdziesiątych 20-tego wieku. Wśród najbardziej znanych wymienić należy: Space invaders, Pac-Man, Galaga.



Rysunek 1.2: Jedna z najbardziej rozpoznawalnych gier zręcznościowych: Pacman.

1.3 Czym jest strategia gracza?

Strategia stanowi zamysł działania gracza. Czasami mylona jest ona z ruchem gracza, gdzie podejmuje on działanie w danej, konkretnej sytuacji. Strategia opisuje działanie dla wszystkich możliwych sytuacji - stanów gry. Można ją przyrównać do algorytmu, co znaczy, że znając strategię danego gracza można wykonać ruch za niego w przypadku jego absencji. Dodatkowo wyróżnić możemy profil strategii, czyli zbiór w którym elementy stanowią poszczególne strategie, które można przypisać poszczególnym graczom. Dzięki strategii stworzę bota, który będzie potrafił odnaleźć się w stworzonej przeze mnie grze. Utworzony bot będzie musiał pokazać, że jest w stanie kierować się pewną polityką, poprzez którą będzie w stanie uzyskać sukces w zaimplementowanej grze. Sukcesem będzie przejście gry z uzyskanym możliwie jak najlepszym wynikiem. Tak stworzonego bota podda się eksperymentom i sprawdzi się czy strategia, którą Wykorzystuje jest optymalna.

Rysunek 1.3: Galaga – gra zręcznościowa typu strzelanka.

2. Implementacja gry

2.1 Wybór środowiska implementacji gry

Postanowiłem stworzyć prostą grę 2D, którą w przyszłości mógłbym rozszerzać albo zmieniać, aby pokazać jak działa uczenie ze wzmocnieniem. Postanowiłem, że stworzę grę w języku C++ w środowisku Microsoft Visual Studio. Są inne języki, które posiadają rozbudowane biblioteki do uczenia ze wzmocnieniem. Dla przykładu Python posiada bibliotekę OpenAi Gym, która dostarcza wszystkich informacji potrzebnych w uczeniu przez co zagadnienie staje się bardzo proste[11]. Ja jednak w swojej pracy musiałem napisać algorytm uczenia sam od podstaw, więc żaden język programowania nie dawał mi przewagi w pisaniu tego typu kodu. Wybrałem więc z tego właśnie powodu język C++, ponieważ jest to język jeden z najlepszych do pisania gier. Posiada on dodatkowo, w miarę prostą w obsłudze bibliotekę do pisania gier 2D SFML.

Rysunek 2.1: Logo biblioteki SFML

2.2 Biblioteka SFML

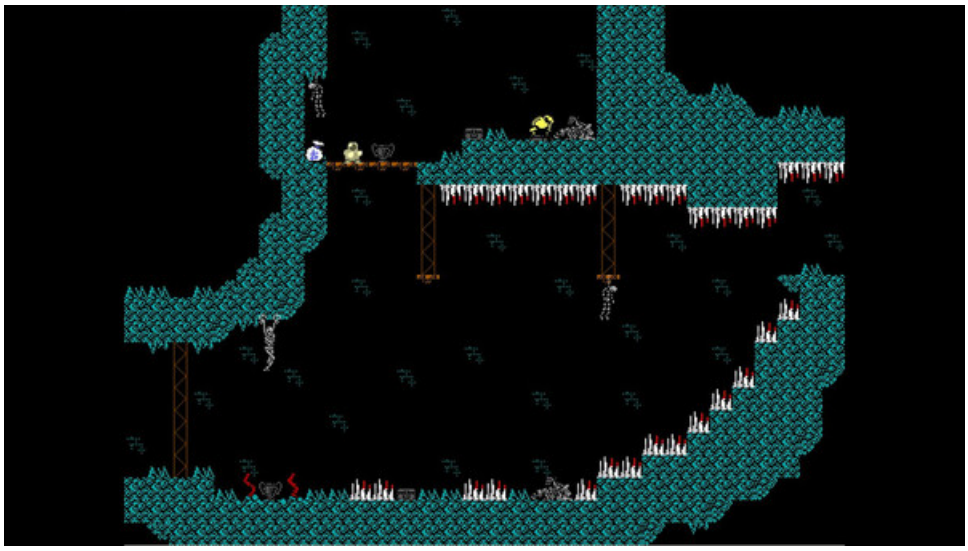
Biblioteka SFML jest to biblioteka, która zapewnia łatwy w obsłudze interfejs do tworzenia gier i wszelakiego rodzaju prezentacji multimedialnych. Można używać jej na wielu systemach operacyjnych i wielu językach programowania. Dzięki niej można tworzyć proste gry typu 2D[12]. Składa się z 5 głównych modułów którymi są:

- Moduł systemowy – odpowiedzialny jest za obsługę wątków i który jest dodatkowo głównym modułem tejże biblioteki,
- Moduł graficzny – pozwala tworzyć, modyfikować i wyświetlać obiekty i teksty,
- Moduł audio – odpowiada za nagrywanie i odtwarzanie dźwięku,
- Moduł sieciowy – umożliwia on tworzenie gier sieciowych,
- Moduł okna – odpowiada za tworzenie okna i wychwytywanie wszystkich zdarzeń wokół niego.

```
1  $include <SFML/Graphics.hpp>
2
3  int main()
4  {
5      sf::RenderWindow window(sf::VideoMode(1920, 1680),
           "Game");
6
7      while (window.isOpen())
8      {
9          sf::Event event;
10         while (window.pollEvent(event))
11         {
12             if (event.type == sf::Event::Closed)
13                 window.close();
14         }
15
16         window.clear();
17         window.display();
18     }
19
20     return 0;
21 }
```

Kod źródłowy 2.1: Fragment kodu wykorzystujący bibliotekę SFML.

Powyższy kod pokazuje jak łatwo stworzyć działające okienko przy pomocy biblioteki SFML. Przy pomocy klasy RenderWindow tworzy nam okienko o odpowiednich wymiarach podanych przez użytkownika. Główna pętla zbiera zdarzenia, które dzieją się w obrębie okna i użytkownik może je na swoje potrzeby odpowiednio oprogramować. Na koniec pętli podaje się najczęściej elementy, które użytkownik chce wyświetlić[13].

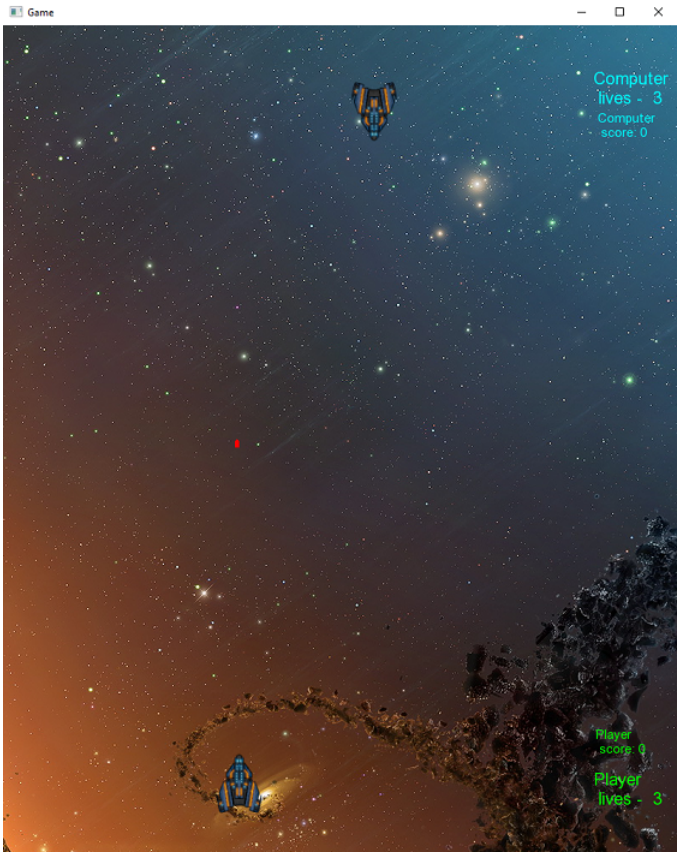


Rysunek 2.2: Gra ”Dispersio” wykonana przy pomocy biblioteki SFML.

W swojej pracy dyplomowej będę używał modułu systemowego, okna i graficznego. Ten ostatni dostarcza szereg klas, które pomogły mi w tworzeniu oprawy wizualnej. Biblioteka ta nie jest bardzo popularna, dlatego większość gier, które są w niej tworzone są głównie małymi projektami. Z tego powodu wybrałem więc SFML, ponieważ jest on wystarczający na potrzeby napisania prostej gry zręcznościowej.

2.3 Gra i jej zasady.

Gra jest prostą platformówką wzorowaną na starej grze *Space Invaders* z lat 70 . Dwa statki umieszczone są na planszy na przeciw siebie w stałej odległości. Ruch statków jest jednak ograniczony poprzez poruszanie się w jednej linii. Mogą się poruszać tylko po osi x, ruch po wszystkich innych osiach jest niemożliwy. Każdy statek ma też możliwość oddania strzału w kierunku przeciwnika. Zadaniem gracza jest zestrzelenie przeciwnika, który posiada 3 życia. Musi jednak uważać, bo przeciwnik nie będzie stał bezczynnie i też może atakować. Grać można zarówno wieloosobowo, jak i z komputerem. Plansza ma wymiary 800 na 1000 i jej wygląd przedstawiono na rysunku 4.1. Do stworzenia zręcznościówki należało stworzyć klasy obiektów statku i pocisków, w których zaimplementowałem wszystkie funkcje umożliwiające poruszanie się i interakcje takie jak: ruch w określonym kierunku, strzał pocisku czy wykrywanie kolizji.



Rysunek 2.3: Planszy gry.

```
    class Ship : public Drawable
{
private:
  Texture s;
  Sprite ship;
  const size_t size{ 40 };
  float shipVelocity{ 80.0f };
  Vector2f velocity{ shipVelocity,0 };
  size_t life{ 3 };
  void draw(RenderTarget& target, RenderStates state)
     const override;

public:
  Ship() = delete;
  Ship(float x, float y,bool w);
  int update(String d, int &dealay);
  int update(int& dealay);
  void shot(vector<Bullet>& bullets, Bullet bullet, int&
     dealay,bool a);
  void shot(vector <Bullet>& bullets, Bullet bullet,
     int& dealay);
```

```
19    float left();
20    float right();
21    int getLife();
22    void resetLife();
23    void setCor(int x, int y);
24    Vector2f getCor();
25    FloatRect getGB();
26    void updatelife(bool x);
27 };
```

Kod źródłowy 2.2: Klasa Ship.

Najważniejszym obiektem gry jest klasa ”Ship”, której zawartość pokazano w kodzie źródłowym 2.2. Klasa dziedziczy po klasie "Drawable", która pomaga wyświetlać obiekty na ekranie dzięki funkcji wirtualnej "draw". Stworzona przeze mnie klasa posiada konstruktor, dzięki któremu możemy stworzyć obiekt podając jego początkowe koordynaty i zmienną typu "bool", która określa kierunek statku. Dodatkowo stworzyłem po 2 funkcje przeciążone ”update” i ”shot”. Jedna z wersji potrzebna będzie do gry w czasie rzeczywistym, a druga do nauki agenta uczenia ze wzmocnieniem. Klasa posiada jeszcze funkcję "getGB", która zwraca kształt obiektu, który przydał się do wykrywania kolizji w grze między pociskiem a statkiem. Pozostałe funkcje służą do kontrolowania ruchu statku w środowisku i modyfikacji jego parametrów.

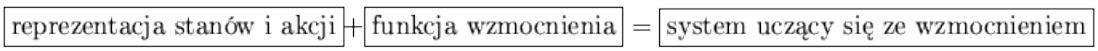
Główna pętla gry działa z uwzględnieniem odświeżania 60 klatek na sekundę. Wychwytuje ona zdarzenia z klawiatury, którymi są odpowiednie klawisze. Gracz może rozpocząć grę z jednym z 3 przeciwników, którymi są: inny gracz, bot losowy i agent uczony Q-Learningiem. W każdej klatce sprawdzane jest, czy któryś ze statków został trafiony. Gracze są informowani o ilości swoich żyć. Dodatkowo na ekranie widoczny jest ”score”, który oznacza liczbę punktów zawodników lub zawodnika i komputera. Zwiększa się on zawsze o 1 na korzyść gracza w przypadku, gdy przeciwnik straci wszystkie swoje życia. Po tym świat jest resetowany i rozpoczyna się kolejna partia.

Rysunek 2.4: Wygrana jednego z graczy.

3. Czym jest uczenie ze wzmocnieniem - omówienie i wybór algorytmu do gry

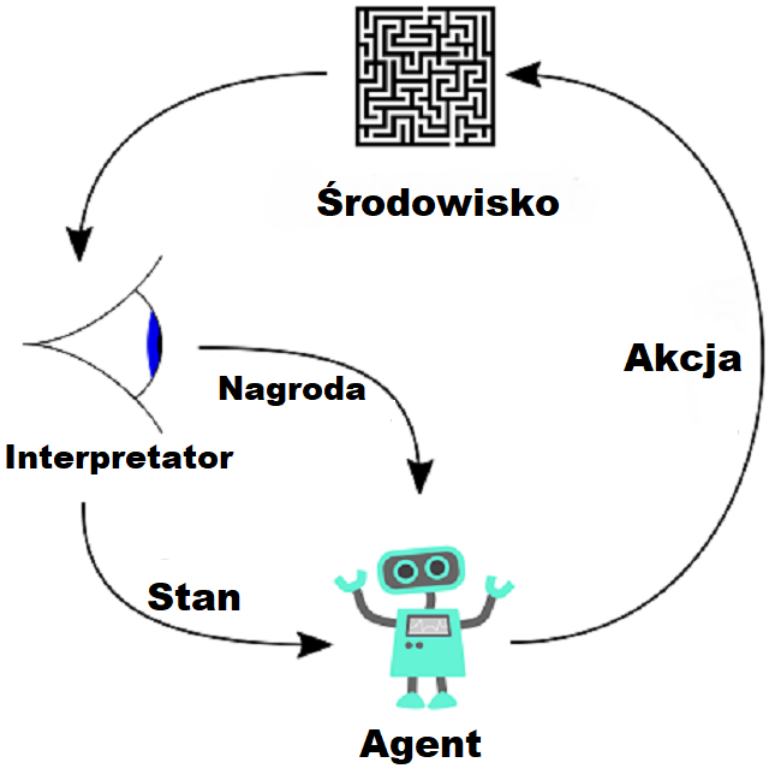
3.1 Uczenie ze wzmocnieniem

Uczenie się ze wzmocnieniem to dziedzina sztucznej inteligencji, a dokładniej uczenia maszynowego, w której agent na podstawie interakcji ze środowiskiem zbiera informacje potrzebne do funkcjonowania w nim. Nasz program poddany takiemu uczeniu nie posiada kompletnej wiedzy o środowisku, w którym eksploruje, w przeciwieństwie do innych dziedzin uczenia maszynowego, gdzie nasz uczony agent wiedzę posiadał od nauczyciela lub na podstawie pewnej bazy pojęć. Ważne jest tylko to aby środowisko było obserwowalne z punktu widzenia agenta. Uczenie ze wzmocnieniem dopuszcza stochastyczność środowiska, czyli stany losowe, co może oznaczać, że w przyszłości agent otrzyma różne nagrody za wykonanie tej samej akcji i co najważniejsze skutkując innym stanem w kolejnym kroku. Zakładamy, że wszystkie zdarzenia pojawiające się w naszym środowisku są nieznane i obce dla naszego ucznia i nie jest w stanie ich zmienić. Oznacza to, że program który uczymy, będzie na początku poruszał się na ślepo w środowisku nieznanym i obcym. Każda akcja wykonana środowisku będzie pociągać za sobą pewne skutki i w zależności od nich, nasz agent będzie otrzymywać nagrodę za swoje działanie lub też karę określoną przez projektanta algorytmu. Na początku będzie podejmował on decyzje, które z punktu obserwatora mogą być nawet absurdalne, ale z czasem będzie posiadał coraz więcej informacji, które będą mówiły jaka akcja w danym stanie będzie najkorzystniejsza. Ogólnie ujmując, agent będzie dążył do maksymalizacji otrzymywanej nagrody i znalezienia przez to optymalnej strategii.[1][2]



Rysunek 3.1: Schemat przedstawiający sposób działania uczenia ze wzmocnieniem.

Algorytm uczenia ze wzmocnieniem można porównać do tresury psa. Na początku szkolenia pies może zachowywać się niegrzecznie i w ogóle nie słuchać naszych poleceń, więc potrzebujemy jakiegoś systemu, w którym będziemy dawać psu nagrody za oczekiwane przez nas zachowania i ewentualne kary, kiedy jego zachowanie będzie złe. Kiedy pies wykona nasze polecanie np. daj łapę (nasza akcja), to otrzyma smakołyk (nagroda). Tak uczone zwierzę z czasem pojmie jakie zachowania będą najkorzystniejsze, aby móc otrzymać upragnioną nagrodę.[5]



Rysunek 3.2: Graf przedstawiający sposób uczenia agenta.

Uczenie ze wzmocnieniem wraz z powiązanym z nim programowaniem dynamicznym wykorzystują procesy decyzyjne Markowa (MDP)[4]. Są one reprezentowane przez stałą czwórkę (X,A,ρ,σ), gdzie:

- X - oznacza skończony zbiór stanów,
- A - oznacza skończony zbiór akcji,
- ρ - oznacza funkcję wzmocnienia,
- σ - oznacza funkcję przejść stanów.

Podsumowując można stwierdzić, że uczenie ze wzmocnieniem jest modelem opartym na interakcjach agenta ze środowiskiem (w naszym przypadku światem gry) w celu odnalezienia optymalnej strategii lub prawie optymalnej. Najważniejszymi elementami takiego uczenia są:

1. Środowisko - miejsce, które nasz uczony agent eksploruje. Poruszanie się właśnie w nim będzie w przyszłości oceniane, bot nie ma możliwości zmieniania praw obowiązujących w środowisku.
2. Agent - uczeń, który w zależności od podejmowanych decyzji będzie odpowiednio wynagradzany lub karany za podejmowane przez siebie akcje.
3. Interpretator/ funkcja oceniająca - będzie zwracać nagrodę w postaci wartości liczbowej za wykonanie akcji w danym stanie. Poprawne określenie tych wartości jest kluczowe, ponieważ musimy wskazać agentowi czego od niego oczekujemy. Złe wyznaczenie funkcji oceniającej może doprowadzić do tego, że otrzymamy wyniki, których nie pożądaliśmy.

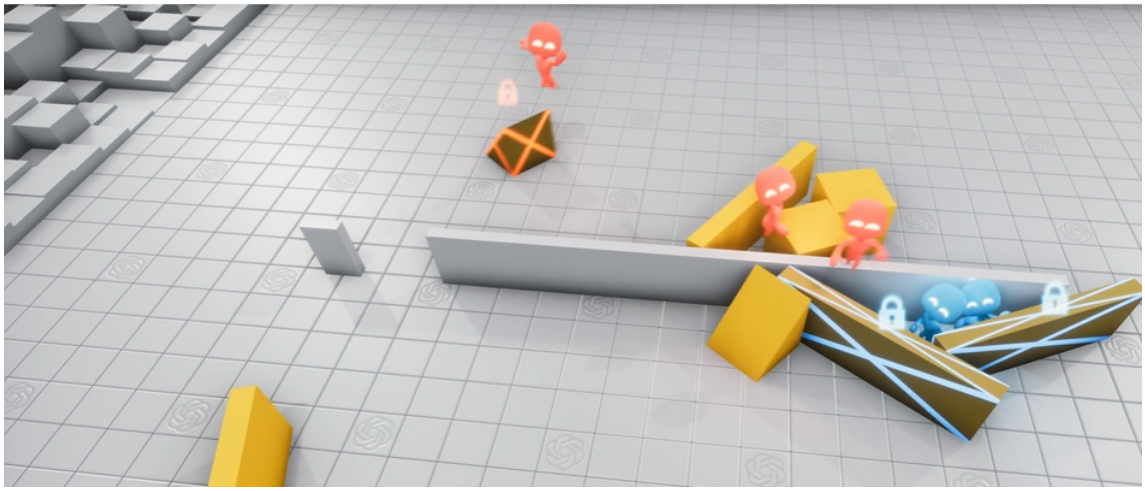
3.2 Do czego wykorzystuje się algorytmy uczenia ze wzmocnieniem.

Uczenie ze wzmocnieniem używamy w:

- W teorii informacji,
- W teorii gier,
- W teorii testowania,
- W statystykach,
- W algorytmach genetycznych.

Najczęściej uczenie ze wzmocnieniem używane jest w różnych rodzajach gier , ponieważ tutaj najczęściej występuje aspekt podejmowania decyzji. Algorytmy te są już tak dopracowane, że często osiągają lepsze wyniki od ludzi. Przez to, że technika jest dosyć prosta w implementacji ma coraz większe zastosowanie. Ostatnio w mediach można było usłyszeć o AlphaZero. Jest to sztuczna inteligencja, który należy do Google. Jest o niej głośno ponieważ z łatwością pokonuje czołowych zawodników w takich grach jak GO czy Szachy. Najciekawszą rzeczą jest jednak, że wygrywa z innymi skomplikowanymi algorytmami, które były mocne w danej grze. W internecie można znaleźć filmy pokazujące jak AlphaZero deklasuje Stockfisha, jeden z najlepszych programów do gry w szachy. Niedawno użyto tej technologii do nauczenia bota w grę DOTA 2. Najlepsi gracze byli pod wrażeniem jego gry i tego że jest praktycznie bezbłędny. To pokazuje, że algorytmy uczenia się ze wzmocnieniem są przyszłościowe i warto bliżej się im przyjrzeć.[6][8]

Ciekawym przykładem jest gra w chowanego, nad którą pracował zespół badawczy z OpenAI. Pracownicy stworzyli grę w której jeden zespół zajmował się chowaniem na planszy a drugi poszukiwaniem ukrytej na mapie drużyny. Na początku zespoły podejmowały proste strategie, potem w miarę zwiększania ilości epizodów taktyki były coraz bardziej rozbudowane i wydawałoby się optymalne. Aż do czasu kiedy uczone zespoły odkryły bugi, czyli błędy występujące w grach i zaczęły wykorzystywać je na swoją korzyść. Pokazuje to jasno jak trudno jest stworzyć środowisko, w którym takie uczenie ze wzmocnieniem może zostać użyte.[7]



Rysunek 3.3: Fragment gry w chowanego.

3.2.1 Wybór algorytmu uczenia ze wzmocnieniem

Najbardziej znane algorytmy uczenia ze wzmocnieniem to AHC, Q-Learning i SARSA. AHC tłumacząc na język polski jest to adaptacyjny heurystyczny krytyk. Składa się z 2 modułów, z których jeden podejmuje decyzję, a drugi podejmuje się oceny tej decyzji. Algorytm musi ten posługiwać strategią, której się uczy. Q-Learning jest algorytmem, który w czasie działania uczy się funkcji wartości akcji. W przeciwieństwa do AHC jest to algorytm, który nie musi stosować strategii, której się uczy. Algorytm SARSA jest zmodyfikowaną wersją algorytmu Q-Learning.

W swojej pracy stanąłem przed wyborem odpowiedniego algorytmu. Przeglądając wszystkie algorytmy najlepszym wyborem okazał się Q-Learning. Jest to algorytm, który cieszy się największą popularnością w ostatnim czasie. Powstało dużo projektów pokazujących wykorzystanie tego algorytmu. Algorytm tworzy tablicę Q, która pokazuje jak bardzo wartościowa jest akcja w danym stanie. W pełni uzupełniona tablica pokazuje uczonemu agentowi jaką decyzję powinien podjąć, aby była ona najkorzystniejsza. Bazuje on na wykonywaniu jednej głównej pętli t razy, w której:

1. Wybieramy akcję i wykonujemy ją.
2. Obserwujemy wzmocnienie r i następny stan.
3. Aktualizujemy wartość dla danego stanu i akcji.

$$\mathbf{Q^{new}}(s_t, a_t) \leftarrow \mathbf{Q}(s_t, a_t) + \alpha * (r_t + \gamma * \max_a \mathbf{Q}(s_{t+1}, a) - \mathbf{Q}(\mathbf{s_t}, \mathbf{a_t})) \tag{3.1}$$

Wzór 3.1 przedstawia jak działa mechanizm aktualizacji w algorytmie Q-Learning. Wykorzystuje on w głównym stopniu równanie Bellmana. Cały algorytm jest aktualizowany w pewnym czasie t. We wzorze:

- $\mathbf{Q^{new}}(s_t, a_t)$ – oznacza wartość Q w stanie s dla akcji a, która w danym kroku będzie analizowana i aktualizowana,
- α jest to współczynnik uczenia, który mieści się w przedziale od 0 do 1 i od niego będzie zależała szybkość uczenia,
- r_t jest to nagroda, która będzie przyznawana za przejście ze stanu s_t do stanu s_{t+1} po wykonaniu akcji a, wartość jej będzie zależała od funkcji wartości akcji, którą zaimplementujemy
- γ nazywana współczynnikiem dyskontowania – z przedziału od 0 do 1 (wartości bliskie 0 premiują nagrody krótkoterminowe, wartości bliskie 1 w sposób fachowy uwzględniają nagrody krótko- i długoterminowe),
- $\max_a \mathbf{Q}(s_{t+1}, a)$ - jest wartością z tabeli Q reprezentującą największą możliwą wartość funkcji Q otrzymywaną dla wszystkich akcji a w stanie s_{t+1}.

Dobór współczynników algorytmu Q-Learning jest bardzo ważny, ponieważ wpływa na cały proces uczenia. W środowiskach nieprzewidywalnych zbyt duży czynnik α może okazać się problemem. Po wykonaniu algorytmu określoną liczbę epizodów t powstanie nam tabela, która posiada optymalną lub zbieżną do optymalnej strategię. W trakcie nauki trzeba zadbać, aby algorytm odwiedził jak największą liczbę stanów, więc algorytm ten często się modyfikuje dodając współczynnik ϵ. Współczynnik ϵ określa prawdopodo-

bieństwo wykonania akcji losowej. Z początku jego wartość powinna być ustawiona tak, aby uczony bot mógł eksplorować jak największą ilość stanów i podejmować losowe akcje w środowisku, a potem ją stopniowo zmniejszać, dzięki czemu agent będzie wykonywał coraz częściej akcje z tabeli Q, co zapewnia pewien kompromis pomiędzy eksploracją a eksploatacją środowiska.[1][10][4]

4. Implementacja agenta wykorzystującego algorytm Q-Learning

Najważniejszym elementem w uczeniu ze wzmocnieniem opartym na algorytmie Q-Learning jest tablica wartości funkcji akcji w danym stanie, która w określonej liczbie cykli będzie tworzona na podstawie nagród przyznawanych w rozegranych grach w środowisku. Stworzyłem do tego klasę QLearning przedstawioną poniżej, w której zawarłem wszystkie potrzebne zmienne i funkcje.

```
1  class QLearning
2  {
3  public:
4    QLearning();
5    ~QLearning();
6    int randomaction();
7    void printQTable();
8    int randomBot(Ship &ship);
9    void useAgent(Ship &computer, Ship &player, vector
       <Bullet>& computerbullets1, Bullet computerbullet1,
       vector <Bullet>& bullets, Bullet bullet, int
       playermove, int &dealay, int& bulletdealay);
10   void trainingAgent(Ship& Ship1, Ship &Ship2);
11   float getReward(Ship& Ship1, Ship& Ship2, vector
       <Bullet>& bullets1, Bullet bullet1, vector
       <Bullet>& bullets2, Bullet bullet2);
12   void testAgent(Ship& Ship1, Ship& Ship2);
13   void greedyTacticVSBot(Ship& Ship1, Ship& Ship2);
14   void greedyTacticVSQL(Ship& Ship1, Ship& Ship2);
15   void botvsbot(Ship& Ship1, Ship& Ship2);
16   void botQLVSbotQL(Ship& Ship1, Ship& Ship2);
17   void updatealfa(float a);
18   void updategamma(float a);
19   void resetQTable();
20 private:
21   float********** QTab;
22   float gamma = 0.6;
23   float alfa = 0.1;
24   }
```

Kod źródłowy 4.1: Klasa Q-Learning.

Klasa posiada wartości współczynników gamma, alfa, które będą potrzebne w uczeniu. Zawiera też liczne funkcje, z których:

- randomaction() - losuje i zwraca losową akcję,
- printQTable() - wypisuje tablicę QTable,
- randomBot() - jest to bot, który wykonuje losową akcję obiektem Ship i zwraca wartość, którą jest ruch tego bota,
- useAgent() - jest o funkcja, która będzie operowała na tablicy QTable i przez nią będzie podejmowała akcję, która będzie najlepsza w danej chwili,
- trainingAgent () - najważniejsza funkcja to w niej będzie uczony bot wykorzystujący Q-Learning,
- getReward() - będzie zwracać nagrodę przyznawaną uczniowi,
- testAgent() - funkcja, która przeprowadzi pojedynek bota losowego z botem Q-Learning,
- greedyTacticVSBot() - umożliwi sprawdzenie gry bota losowego i bota z taktyką zachłanną,
- greedyTacitcVSQL() - umożliwi sprawdzenie gry bota Q-Learning i bota z taktyką zachłanną,
- botvsbot() - będzie to przypadek testowy i będzie polegał na pojedynku 2 botów losowych,
- botQLVSbotQL() - uczony bot zagra sam ze sobą,
- updatealfa() - funkcja aktualizująca współczynnik alfa,
- updategamma() - funkcja aktualizująca współczynnik gamma,
- resetQTable() - przywraca tabelę QTab do stanu początkowego.

4.1 Przystosowanie gry do implementacji tabeli Q-Learning.

Już na początku implementacji algorytmu Q-Learning stanął przed problemami. Musiałem dokładnie określić stany i akcje, na których uczony bot będzie operował podczas nauki. Pierwszym i drugim elementem składającym się na stan było oczywiście położenie statków, jak wiadomo statki mogły poruszać się tylko po osi x i było to ograniczone światem gry. Wynikałoby z tego, że położenie moich obiektów już tworzy liczbę stanów, która wynosiłaby 800*800=640 000. Nie można było na to pozwolić, więc podzieliłem planszę na dyskretne stany o stałej określonej długości, których liczba wynosiła 10. Statki poruszały się ze stałą prędkością, więc w bardzo łatwy sposób można było tego dokonać. Na potrzeby gry zredukowałem również możliwe pozycje kul poprzez ich dyskretyzacje. Pozwoliło to na znaczne zredukowanie wielkości przyszłej tablicy wartości funkcji. Kolejnymi informacji, które dostarczam to prędkość statku przeciwnika w postaci: ruchu w lewo, w prawo i bezczynności. Kolejnymi stanami były 4 informacje o pociskach przeciwnika i jedna o własnym pocisku. Finalnie tablica używana w Q-Learningu będzie posiadała takie stany jak:

1. pozycja własna statku - określać będzie sektor gracza na osi x w postaci dyskretnych stanów od 0 do 9,
2. pozycja statku przeciwnika - tak jak wyżej, określać będzie sektor przeciwnika na

osi x w postaci dyskretnych stanów od 0 do 9,
3. prędkość/ruch przeciwnika - będzie reprezentować ruch przeciwnika (0,1,2), gdy wartość ta będzie 0 to przeciwnik stoi bezczynnie albo strzela, wartości 1 i 2 będą kolejno oznaczać ruch w lewo albo prawo,
4. pozycja kuli przeciwnika w osi x - tak samo jako w pozycji statku, pozycje kuli w osi x będzie reprezentować wartość dyskretna z przedziału od 0 do 9,
5. czy kula przeciwnika leci w kierunku własnego statku - wartość typu "bool", będzie pokazywać czy tor lotu przeciwnika przecina pozycje statku,
6. czy kula przeciwnika znajduje się w sektorze od 900 do 700 (wartość 0 albo 1),
7. czy kula przeciwnika znajduje się w sektorze od 700 do 400 (wartość 0 albo 1),
8. czy kula przeciwnika znajduje się w sektorze od 400 do 100 (wartość 0 albo 1),
9. czy własna kula leci w kierunku przeciwnika - ostatnim stanem będzie wartość typu ”bool” informująca czy tor lotu pocisku uczonego agenta przecina pozycje wrogiego statku.

Liczba stanów wyniesie w ten sposób 10*10*3*10*2*2*2*2*2=96 000. Akcje, które będą dostępne w środowisku dla uczonego agenta to: bezczynność, ruch w lewo, ruch w prawo i strzał. Powstanie dzięki temu tablica stanów i akcji QTab(S,A), o finalnej liczbie stanów 384 000.

4.1.1 Algorytm Q-Learning

Algorytm Q-Learaning został napisany bez używania jakichkolwiek bibliotek pomocniczych. Omówię tylko najważniejsze elementy algorytmu, który zaimplementowałem. Algorytm uczenia ze wzmocnieniem ropoczyna się od ustalenia pozycji obu statków i przypisania ich do zmiennych “current_pos1“ i “current_pos2“. Następnie następuje wykonanie ruchu przez bota losowego, który przy pomocy funkcji randomaction() losuje jedną z dostępnych akcji. Jeśli jej numer wynosi 0 to pozycja przeciwnika się nie zmienia się i zmienna “next_pos2” równa się zmiennej “current_pos2“, w przypadku wylosowania pierwszej i drugiej pozycja statku jest aktualizowana odpowiednio w lewą lub prawą stronę i zmienna “next_pos2“ jest odpowiednio odejmowana lub dodawana o 1. W przypadku wylosowania ostatniej akcji zmienna ”next_pos2” równa się ”current_pos” i wykonywany jest strzał. Dodatkowo funkcja randomBot() zwraca ruch tego bota. W kolejnych krokach ustalane są stany najpóźniej wystrzelonej kuli, zarówno uczonego agenta jak i jego przeciwnika, z czego tylko jeden stan dotyczy jego własnej, co zostało przedstawione w algorytmie 4.2.

```
1 if (!bullets1.empty())
2   {
3     if (abs(Ship2.getCor().x - bullets1.front().get().x)
        < 40)
4     {
5       mb = 1;
6     }
7   }
8
9 if (!bullets2.empty())
```

```
10    {
11    bc = bullets2.front().get().x / 80;
12
13    if (abs(Ship1.getCor().x - bullets2.front().get().x) <
          40)
14    {
15      bo1 = 1;
16    }
17    if (bullets2.front().get().y < 900 &&
          bullets2.front().get().y > 700)
18    {
19      so1 = 1;
20    }
21    if (bullets2.front().get().y <= 700 &&
          bullets2.front().get().y > 400)
22    {
23      so2 = 1;
24    }
25    if (bullets2.front().get().y <= 400 &&
          bullets2.front().get().y > 100)
26    {
27      so3 = 1;
28    }
29  }
```

Kod źródłowy 4.2: Ustalanie stanów kul potrzebnych do algorytmu.

Następnie sprawdzamy czy kontenery typu “vector“ o nazwach “bullets1“ i “bullets“ są puste. Jeśli nie, to pobierane są wszystkie informacje o kulach. Pociski bota losowego są aktualizowane.

Dzięki zgromadzonym informacjom mamy pełną wiedzę o wszystkich stanach. Ustawiamy zmienną pomocniczą “max“ na wartość odpowiadającej tym stanom i akcji bezczynność. W kolejnym kroku losowana jest liczba z przedziału od 0 do 1 i porównywana ona jest ze zmienną ”epsilon“. Jeśli jest ona większa od niej to wybierana jest najlepsza akcja dla podanych stanów z tablicy ”QTab”, w przeciwnym wypadku losowana jest akcja, którą wykona uczeń. Tak samo jak w bocie opartym o losowość wszystkie stany o własnej pozycji i kulach są aktualizowane w świecie gry. Wszystkie te kroki zostały zaprezentowane w kodzie źródłowym 4.3.

```
1  max =
       QTab[current_pos1][current_pos2][move][bc][bo1][so1]
2       [so2][so3][mb][0];
3  if ((float)rand() / RAND_MAX >= epsilon)
4  {
5    for (size_t z = 0; z < actions; z++)
6    {
7      if (QTab[current_pos1][current_pos2][move][bc][bo1]
8      [so1][so2][so3][mb][z] > max)
9      {
```

```
10          max =
                QTab[current_pos1][current_pos2][move][bc][bo1]
11          [so1][so2][so3][mb][z];
12          ra1 = z;
13      }
14    }
15  }
```

Kod źródłowy 4.3: Wybór pomiędzy eksploatacją a eksploracją środowiska i przyznawanie nagrody.

Kluczowym i następnym w kolei krokiem było ocenienie pozycji i przyznanie odpowiedniej nagrody naszemu uczonemu agentowi. Wywołano funkcję ”getReward”, która w argumentach przyjmuje oceniany statek, jego przeciwnika i odpowiednie dane przynależnych im kul. Po pierwsze sprawdza ona, czy jakikolwiek pocisk naszego uczonego agenta trafił we wrogi statek, dzięki zaimplementowanej w klasie ”Bullet” funkcji informującej o kolizji z obiektem, która jako argument przyjmuje współrzędne kształtu obiektu. Jeśli tak to jest przyznawana nagroda w postaci 1 punktu, a trafionej jednostce odejmowane jest życie. Dodatkowo jeśli życia przeciwnika będzie równe zero, czyli zostanie spełniony warunek wygrania gry, to uczony obiekt otrzymuje nagrodę równą 100. Analogiczne sprawdzamy, czy wrogie kule nie trafiły naszego bota, odejmując 1 punkt nagrody za trafienie go pociskiem, ale za przegranie odejmujemy już 10. Do ”reward” przypisywana jest zwracana wartość funkcji, do ”wreward” nagroda za jeden epizod, a do ”globalreward” cała nagroda w procesie uczenia. Wszystko zostało przedstawione w kodzie źródłowym poniżej.

```
1  reward = getReward(Ship1, Ship2, bullets1, bullet1,
       bullets2, bullet2);
2  wreward = wreward + reward;
3  globalreward = globalreward + reward;
4
5  float QLearning::getReward(Ship& Ship1, Ship& Ship2,
       vector<Bullet>& bullets1, Bullet bullet1, vector
       <Bullet>& bullets2, Bullet bullet2)
6  {
7  float reward = 0;
8  if (!bullets1.empty())
9  {
10   for (size_t i = 0; i < bullets1.size(); i++)
11   {
12     if (bullets1[i].checkCollision(Ship2.getGB()))
13     {
14       Ship2.updatelife(0);
15       reward += 1;
16       if (Ship2.getLife() == 0)
17       {
18         reward += 100;
19       }
20       bullets1.erase(bullets1.begin() + i);
```

```
        break;
      }
    }
  }
  if (!bullets2.empty())
  {
    for (size_t i = 0; i < bullets2.size(); i++)
    {
      if (bullets2[i].checkCollision(Ship1.getGB()))
      {
        Ship1.updatelife(0);
        reward -= 1;
        if (Ship1.getLife() == 0)
        {
          reward -= 10;
        }
        bullets2.erase(bullets2.begin() + i);
        break;
      }
    }
  }
    return reward;
}
```

Kod źródłowy 4.4: Przyznawanie nagrody agentami za wykonane akcje.

Następnie do ”max” przypisujemy możliwą największą wartość ze wszystkich akcji. Po wszystkich tych krokach przyszedł czas na najważniejszą część algorytmu Q-Learning, czyli aktualizacje ”QTab”, której algorytm zamieszczono poniżej. Poprzednia wartość jest tabeli nadpisywana nową wartością, wyliczoną ze wzoru 3.1.

```
max =
   QTab[next_pos1][next_pos2][move][bc1][bn1][sn1][sn2]
    [sn3][mb1][0];
for (size_t z = 0; z < actions; z++)
  {
    if (QTab[next_pos1][next_pos2][move][bc1][bn1][sn1]
    [sn2][sn3][mb1][z] > max1)
    {
      max1 =
         QTab[next_pos1][next_pos2][move][bc1][bn1][sn1]
      [sn2][sn3][mb1][z];
    }
  }

QTab[current_pos1][current_pos2][t2][bc][bo1][so1][so2]
[so3][mb][ra1] =
   QTab[current_pos1][current_pos2][t2][bc][bo1][so1][so2]
[so3][mb][ra1] + alfa * (reward + gamma * max -
   QTab[current_pos1][current_pos2][t2][bc][bo1][so1][so2]
```

```
16 [so3][mb][ra1];
```

Kod źródłowy 4.5: Aktualizacja tabeli Q.

Po wszystkich tych operacjach sprawdzany jest warunek wygrania lub przegrania gry. W takim przypadku dla jednego ze statków zwiększany jest licznik wygranych. Na ekranie jest wypisywana nagroda za ten epizod, elementy świata są resetowane. Aktualizowana jest także zmienna epsilon zgodnie z regułą epsilon=epsilon - 1 / liczba epizodów do rozegrania i wszystkie zmienne pomocnicze są resetowane. Jeśli jednak ten warunek nie jest spełniony to zmienne ”current_pos1” i ”current_pos2” są nadpisywane odpowiednio zmiennymi ”next_pos1” i ”current_pos2”, a zmienne pomocnicze są resetowane. Tak sprecyzowany algorytm działa, aż do momentu, w którym rozegra określoną liczbę epizodów. Dzięki całej nauce otrzymano tabelę, której fragment można zobaczyć na rysunku poniżej.

```
0 0 0 0
0 0.53587 -0.00186951 0.251223
0 0 -0.01 0
0.171239 -0.599215 0.200529 0.14965
0.329259 13.6932 6.77271 6.1588
0.429307 0.432196 0.383208 0.426149
0 0.797209 5.49898 2.47013
0 0 0 0
0 0 0 0
0.20075 0.468223 0.122507 0.0707541
-0.019 1.21099 -0.01 0
```

Rysunek 4.1: Fragment otrzymanej tabeli Q.

4.2 Bot losowy i algorytm zachłanny

Na potrzeby pracy trzeba było stworzyć innych agentów, aby móc porównać działanie algorytmu Q-Learning. Bot losowy używany był do nauki bota z algorytmem Q-Learning. Jego główną logiką jest po prostu losowe wybranie jednej z 4 możliwych opcji, którymi są: strzał, ruch w lewo, ruch w prawo i bezczynność. Drugim agentem, który został zaimplementowany był bot posługujący się algorytmem zachłannym, czyli takim który podejmuje decyzje tylko lokalnie optymalne, czyli najlepsze w danym momencie. Kierował się on następującymi instrukcjami:

- kiedy nie ma kuli przeciwnika w linii toru lotu i własny statek jest naprzeciwko przeciwnika to oddaj strzał,
- kiedy nie ma kuli przeciwnika w linii toru lotu i własny statek nie jest naprzeciwko przeciwnika, to dopóki nie będzie naprzeciwko niego, to wykonuj ruch w jego stronę,

- kiedy kula leci w kierunku statku, wykonaj ruch w lewo albo w prawo, chyba że znajdujesz się w skrajnych miejscach planszy to wykonaj ruch do środka planszy,
- kiedy jesteś naprzeciwko statku i kula leci w kierunku własnego statku ale w odległości większej niż 200 to oddaj strzał,
- kiedy jesteś naprzeciwko statku i kula leci w kierunku własnego statku ale w odległości mniejszej niż 200 to wykonuj ruch w lewo albo w prawo, chyba że znajdujesz się w skrajnych miejscach planszy to wykonaj ruch do środka planszy.

Bota losowego i agenta opierającego się na algorytmie zachłannym użyłem do stworzenia porównań i przetestowania Q-Learningu pod względem wygrywanych gier, jak i uzyskiwanych średnich nagród.

5. Badania i porównanie agentów

5.1 Porównanie zaimplementowanych agentów

Aby móc porównać agenta QL trzeba było na początku stwierdzić, ile gier musi rozegrać agent, by osiągać nagrodę i wyniki na poziomie zadowalającym, czyli takim w którym uczony bot wygrywa prawie wszystkie rozegrane partie. Stworzyłem więc zestawienie 5.1 pokazujące liczbę wygranych bota QL na sto rozegranych gier z botem losowym i średnią nagrodą w zależności od jednego pełnego epizodu gry.

Tabela 5.1: Liczba wygrywanych partii przez agenta i jego średnia nagroda zależnie od liczby rozegranych gier.

Liczba rozegranych gier w nauce	10	100	250	500	1000	10000	20000
Bot QL	8	36	48	84	94	99	99
Bot losowy	92	64	52	16	6	1	1
Średnia nagroda bota QL	-47.41	-10.74	21.22	55.60	60.59	63.65	64.13

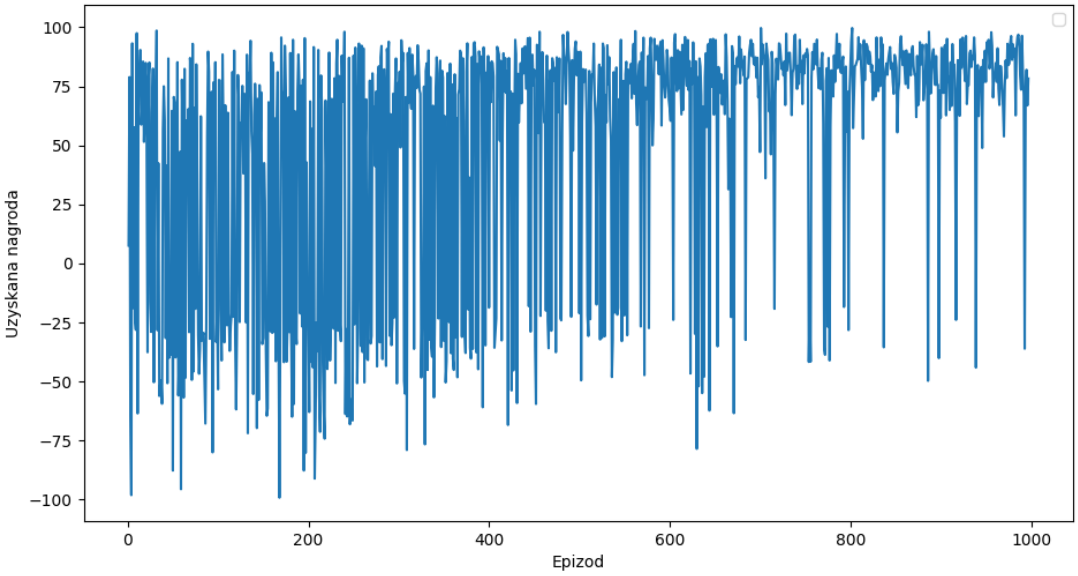
Z przedstawionego zestawienia wynikło, że liczba gier poniżej 200 była zdecydowanie za mała. Na poziomie 250 stosunek wygranych do przegranych wynosił już 50%, a liczba epizodów na poziomie 10 000 i wzwyż nie podnosiła już wygrywalności i otrzymwanej średniej nagrody bota QL. Tabele 5.1 przedstawiono dla wartości:

1. alfa=0.1
2. gamma=0.6

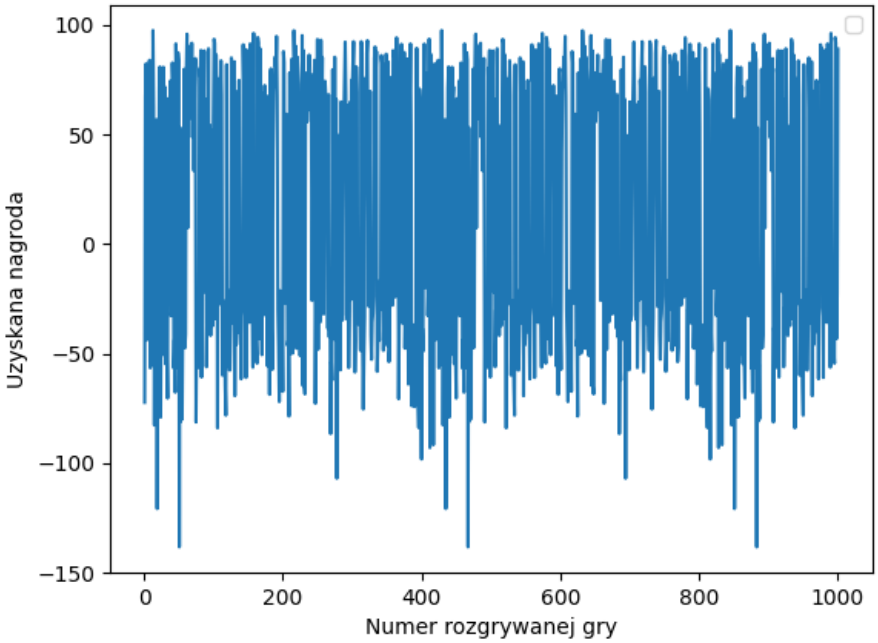
Dalsze testy bota operującego na algorytmie uczenia ze wzmocnieniem zostały przeprowadzone dla liczby rozegranych gier równej 1 000. Tak wybrana liczba gwarantowała, że poziom nauki będzie wysoki. Na tej podstawie stworzono wykres 5.1. Pokazywał on pojedynek bota losowego z agentem Q-Learning. Widać na nim że, mniej więcej od połowy wyznaczonych epizodów zaczął częściej zdobywać większe nagrody i wygrywać. Wiąże się to z wartością epsilon, która służyła po to, aby nasz uczeń wystarczająco eksplorował środowisko. Od tego momentu wynosiła pół swojej wartości. Można zaobserwować też że bot wciąż popełniał błędy na końcowym etapie uczenia, ale było ich mniej niż w początkowym etapie.

Przypadkiem testowym oczywiście był bot losowy walczący sam ze sobą. Na wykresie 5.2 można zauważyć, że bot potrafił osiągać skrajnie niskie wyniki i wraz z liczbą rozgrywanych gier ich jakość się nie poprawiała.

W kolejnym eksperymencie postanowiłem porównać pojedynek agentów ze sobą i wyłonić najlepszego z nich. Parametry bota uczonego algorytmem Q-Learning były takie



Rysunek 5.1: Wykres przedstawiający nagrodę agenta w procesie uczenia.



Rysunek 5.2: Wykres przedstawiający nagrodę bota losowego w trakcie gry sam ze sobą.

same jak w poprzednim eksperymencie. Każda para agentów miała do rozegrania 100 gier. Wyniki tej rywalizacji zamieszczono w tabeli 5.2. Algorytm Q-Learning okazał się najsilniejszy. Pokonał on bota losowego w stosunku 99:1, a algorytm zachłannym

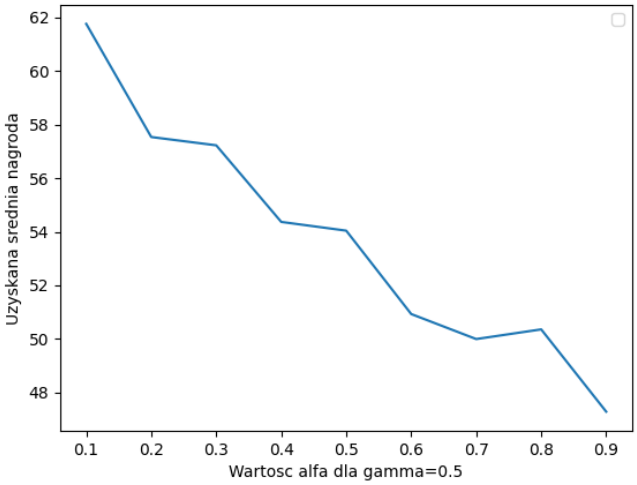
87:13. Widać że, mimo wygrywalności bot wciąż nie jest doskonały i posiada pewne mankamenty. Algorytm zachłanny całkiem dobrze grał przeciwko agentowi losowemu. Jeden z eksperymentów się nie powiódł, ponieważ uczenie ze wzmocnieniem stosowało, w niektórych momentach taktykę wyczekiwania na ruch przeciwnika, dlatego że uważało że lepiej w danym stanie zachować bezczynność lub oddawać strzał w kierunku przeciwnika, aż sam w niego wejdzie. To doprowadziło do tego, że test nigdy się nie kończył.

Tabela 5.2: Tabela przedstawiająca pojedynki między agentami.

Agent 1	Agent 2	Liczba wygranych gracza 1	Liczba wygranych gracza 2
Bot QL	**Bot losowy**	99	1
Bot QL	**Algorytm zachłanny**	87	13
Bot QL	**Bot QL**	Eksperyment nieudany	
Bot losowy	**Bot losowy**	52	48
Bot losowy	**Algorytm zachłanny**	8	92

5.2 Dobór najlepszych współczynników w algorytmie

W algorytmie Q-Learning najważniejszym elementem w trakcie uczenia jest aktualizacja tabeli wartości akcji. Jest ona zależna od dobieranych współczynników alfa i gamma. Analiza ich została przedstawiona poniżej.

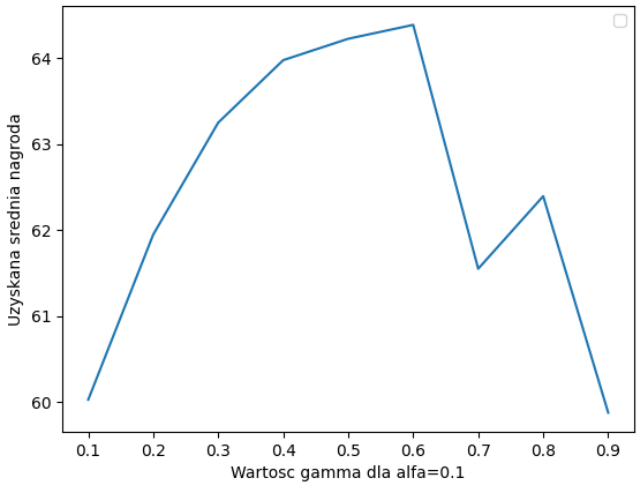


Rysunek 5.3: Wykres przedstawiający średnią nagrodę za rozegraną grę w zależności od współczynnika alfa.

Tabela 5.3: Tabelka przedstawiająca średnią nagrodę zdobywaną przez agenta zależnie od współczynnika alfa

Alfa	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Średnia nagroda	61.76	57.54	57.23	54.37	54.04	50.93	49.99	48.92	47.28

Pierwszym współczynnikiem, który brałem pod uwagę był parametr alfa. Eksperyment przeprowadzono gdy gamma=0.5. Odpowiada on za szybkość uczenia algorytmu Q-Learning. Na wykresie 5.3 i w tabeli 5.3 dokładnie widać, że najlepszą wartością współczynnika alfa jest 0.1, którą zastosowano przez to w projekcie. Można zauważyć też tendencję spadkową otrzymywanych nagród wraz ze zwiększaniem go w stronę wartości 1.



Rysunek 5.4: Wykres przedstawiający średnią nagrodę za rozegraną grę w zależności od współczynnika gamma.

Tabela 5.4: Tabelka przedstawiająca średnią nagrodę zdobywaną przez agenta zależnie od współczynnika gamma

Gamma	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Średnia nagroda	60.02	61.94	63.25	63.97	64.22	64.38	61.55	62.39	59.87

Drugim rozpatrywanym parametrem był współczynnik gamma, kiedy współczynnik alfa wynosił 0.1. Na podstawie wykresu 5.4 i tabeli 5.4 widać, że przy współczynniku równym 0.6 agent osiągał największą średnią nagrodę. Zbliżanie tej wartości w stronę 0 albo 1 powodowało, że funkcja wartości akcji dawała mniejsze wyniki.

5.3 Wnioski

Stworzony agent bazujący na algorytmie Q-Learning osiągał zadowalające wyniki i potrafił wygrywać, dzięki nauczonej strategii. Współczynniki zostały tak dobrane, aby osiągał on jak najlepsze rezultaty. W trakcie rywalizacji z botem losowym i z algorytmem zachłannym okazał się zdecydowanie lepszy. Grając z agentem Q-Learning musiałem się bardzo skupić aby wygrać. Bot w czasie rozgrywki starał się być blisko mojego statku, ale zawsze gdy stałem z nim na linii strzału to oddawał strzał i uciekał w jedną ze stron. Nawet osaczony przez kule na końcu planszy potrafił z tego wybrnąć.

Podsumowanie

Celem niniejszej pracy dyplomowej było stworzenie prostej gry zręcznościowej oraz stworzenie agenta zdolnego do odnajdowania strategii w grze. Dzięki środowisku Microsoft Visual Studio i SFML udało stworzyć się grę platformową, która umożliwiała zaimplementowanie jednego z algorytmów uczenia ze wzmocnieniem. W trakcie wyboru odpowiedniego algorytmu uznałem, że najlepiej spełni swoją funkcję algorytm zwany Q-Learning.

Stworzono klasy i funkcje umożliwiające zademonstrowanie działania wybranego uczenia ze wzmocnieniem. Utworzono między innymi bota losowego, dzięki któremu możliwe było uczenie naszego bota. Wiele funkcji zaimplementowanych w klasie "QLearning" pomogło stworzyć porównania pomiędzy wybranym algorytmem a botami, z których jeden był losowy,a drugi kierował się strategią zachłanną. Udało stworzyć się agenta, który potrafił odnaleźć się w środowisku gry i zdobywać w nim wysokie nagrody. Potrzeba było do tego rozegrać bardzo dużą liczbą gier ze względu na duży rozmiar środowiska.

Testowanie pokazało, że otrzymany agent zdecydowanie przewyższał inne boty zarówno wygrywając z nimi większość rozgrywanych partii jak i zdobywając zdecydowanie lepsze nagrody. Dzięki analizie głównych współczynników algorytmu Q-Learning udało się ustalić, że uczenie ze wzmocnieniem działało najlepiej, kiedy parametr gamma wynosił 0.6 , a alfa 0.1. Wyniki pokazały, że wartość współczynnika alfa, pokrywa się z rekomendowaną wartością w tym algorytmie odnajdywaną w wielu źródłach.

W trakcie tworzenia bota opartego na uczeniu ze wzmocnieniem napotkałem się na wiele trudności. Pierwszym z nich było środowisko gry. W algorytmach opartych na tym systemie uczenia trzeba bardzo dokładnie zadbać o poprawne określenie stanów i akcji. W przeciwnym razie można otrzymać niepożądane wyniki. Drugim problemem było ustawienie parametrów, które potrafiły zdecydowanie zmieniać przebieg nauki. Drobna zmiana w systemie nagrody powodowała całkiem inny rezultat. Zaskoczył mnie w pewnym stopniu ostateczny wynik działania algorytmu Q-Learning, ponieważ posługiwał się on inną strategią niż przewidziałem. Myślałem, że bot będzie próbował pociskami zagonić przeciwnika do granicy planszy, ale agent zastosował taktykę wyczekiwania i atakował tak aby przeciwnik wszedł w tor lotu jego kuli, co było w pewnym stopniu nieoczekiwane.

Kolejnym celem mojej pracy mogłoby być większe skomplikowanie gry, w celu odkrywania obserwowania zmian jakie one wprowadzają w uczeniu ze wzmocnieniem. W pracy można byłoby ulepszyć działanie algorytmu poprzez dodanie większej liczby stanów lub jeszcze lepsze dostosowanie środowiska gry. Istnieją też bardziej skomplikowane warianty tego sposobu uczenia takie jak Deep Q-Learning lub Double Q-Learning, które

mogłyby wskazać inne strategie niż te otrzymane lub je usprawnić. Stworzenie różnych wersji uczenia ze wzmocnieniem mogłoby wskazać też różnice między tymi algorytmami i ich zarówno słabe i mocne strony. Jest to na pewno przyszłościowa i ciekawa dziedzina, której rozwijanie może przynieść nowe możliwości.

Spis literatury

Książki

[1] Paweł Cichosz. *Systemy uczące się*. WNT, 2000. ISBN: 83-204-2544-1.

[2] Richard S. Sutton i Andrew G. Barto. *Reinforcement Learning:An Introduction*. The MIT Press, 2018.

Artykuły

[3] Jan Stasieńko. "Gry komputerowe – jestem na „tak", jestem na „nie".Zagrożenia, szanse i wyzwania rozrywki komputerowej". W: (2013).

Źródła internetowe i inne

[4] URL: http://wazniak.mimuw.edu.pl/index.php?title=Sztuczna_inteligencja/SI_Modu%C5%82_13_-_Uczenie_si%C4%99_ze_wzmocnieniem.

[5] *A Beginners Guide to Q-Learning*. URL: https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c.

[6] Ben Dickson. *AI defeated human champions at Dota 2. Here's what we learned.* URL: https://bdtechtalks.com/2019/04/17/openai-five-neural-networks-dota-2/.

[7] *Emergent Tool Use from Multi-Agent Interaction*. URL: https://openai.com/blog/emergent-tool-use/.

[8] *Google's AlphaZero Destroys Stockfish In 100-Game Match*. URL: https://www.chess.com/news/view/google-s-alphazero-destroys-stockfish-in-100-game-match.

[9] *How Many People Play Video Games in The World?* URL: https://weplayholding.com/blog/how-many-people-play-video-games-in-the-world/.

[10] Sayak Paul. *An introduction to Q-Learning: Reinforcement Learning*. URL: https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/.

[11] *Reinforcement Q-Learning from Scratch in Python with OpenAI Gym*. URL: https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/?fbclid=IwAR2_p0Ox19glKnMjMifNC2v1//yhm4O74YV0hZpd-fTOHoRyImPKKjmkP4Jr0.

[12] *SFML Documantation*. URL: https://www.sfml-dev.org/documentation/2.5.1/.

[13] *SFML Tutorials*. URL: https://www.sfml-dev.org/tutorials/2.5/.