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

Kierunek: **Informatyka**
Specjalność: -

Wiktor Wajszczuk
nr albumu: 296534

Zastosowanie uczenia przez wzmacnianie w środowisku agentowym

Application of reinforcement learning in an agent-based environment

Praca licencjacka
napisana w Katedrze Oprogramowania Systemów Informatycznych
pod kierunkiem dr Marcina Denkowskiego

Lublin, rok 2022

Spis treści

Wstęp

Jedną z większych dziedzin uczenia maszynowego jest uczenie przez wzmacnianie (ang. *reinforcement learning*) W odróżnieniu od zarówno uczenia nadzorowanego i nienadzorowanego nie potrzebujemy w tym przypadku żadnych gotowych danych wejściowych i wyjściowych. Zamiast tego, algorytm pozyskuje dane na bieżąco ze środowiska do, którego jest zastosowany. Dzięki temu, że algorytmy uczenia przez wzmacnianie nie mają tego ograniczenia możemy zastosować je do problemów takich jak gra na giełdzie [1], czy nauka grania w gry, w swojej pracy skupię się na tym drugim.

Celem pracy jest zaimplementowanie algorytmu uczenia przez wzmacnianie Q-Learningu. Następnie zoptymalizowaniu go tak by po zastosowaniu go do własnoręcznie zaimplementowanej gry był w stanie osiągnąć w niej możliwie najwyższy wynik. Do zrealizowania tego zostanie użyty język python oraz moduł pygame.

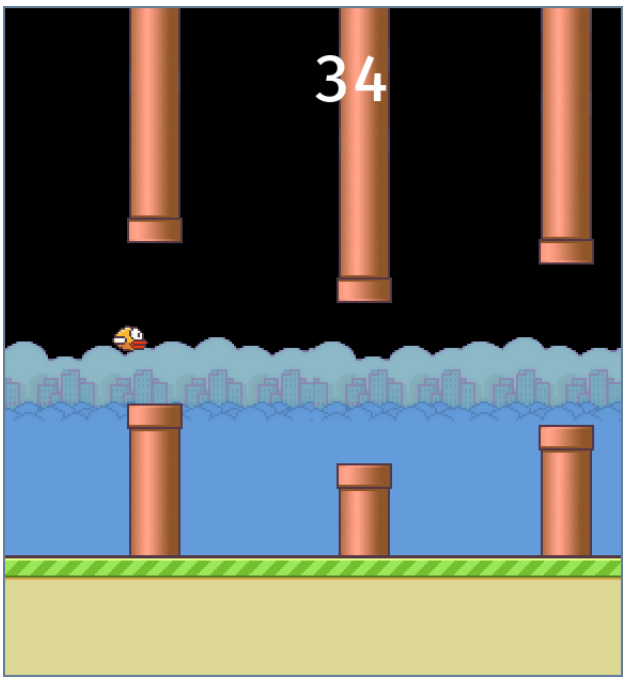
Na początku przedstawię jaką grę wybrałem, opiszę jej zasady oraz zaprezentuję szczegóły jej implementacji. Następnie przybliżę zagadnienie uczenia przez wzmacnianie oraz algorytmu Q-learning. Pod koniec pokażę jak zaimplementowałem wspomniany algorytm, oraz testy po optymalizacjach algorytmu, które zastosowałem.

Rozdział 1

Gra

Problem, który postawiłem przed Q-learningiem to nauczenie się grania w “Flappy Bird” – grę z 2013 autorstwa wietnamskiego developera Dong Nguyena [2]. Zdecydowałem się właśnie na tą grę, gdyż sterowanie w niej jest bardzo proste – jest tylko jeden przycisk do kontrolowania toteż jedna akcja do podjęcia przez algorytm.

1.1 Zasady gry



Rysunek 1.1: Zrzut ekranu z gry

Gracz kontroluje obiekt reprezentowany w grze jako żółty ptak, jego zadaniem jest tak nim sterować by omijać obiekty reprezentowane przez czerwone rury oraz nie pozwolić na to by siła grawitacji doprowadziła do kolizji obiektu ptaka z obiektem ziemi (na rysunku

1.1 zielona część na dole). Na obiekt ptaka działa jedynie siła grawitacji a obiekty rur przesuwane są w lewo w jego stronę ze stałą prędkością. Gracz ma możliwość kontrolować jedynie czy obiekt, którym steruje zwiększy swoją wysokość o stałą, zakodowaną ilość jednostek czy tego nie zrobi i obiekt reprezentowany przez ptaka zmniejszy swoją wysokość. Gra kończy się gdy dojdzie do kolizji obiektu ptaka z, którymkolwiek obiektem rur – dolnej bądź górnej albo z obiektem ziemi. Punkty przyznawane są jeżeli graczowi uda się ominąć nadchodzące przeszkody przelatując przez szczelinę między nimi. Za każde ominięcie przyznawany jest jeden punkt. Na górze rysunku 1.1 białymi cyframi wypisana jest ilość osiągniętych przez gracza punktów. Jak widać gracz ominął już trzydzieści cztery pary rur. Podsumowując – gra sprowadza się do kontrolowania wysokości ptaka tak by ten przelatywał między nadchodzącymi rurami.

1.2 Wykorzystane technologie

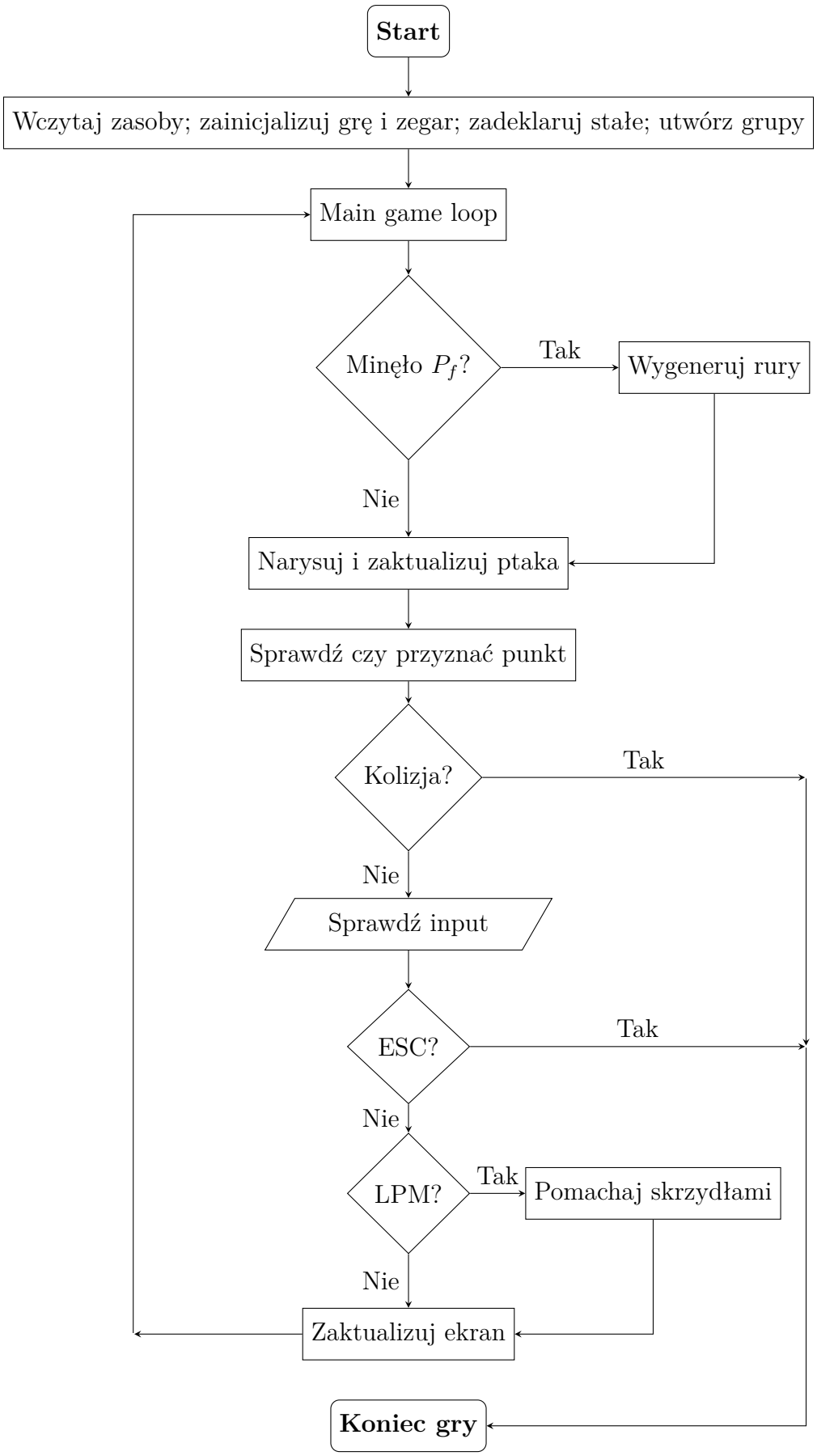
Do zaimplementowania tej gry zdecydowałem się na użycie wysoko poziomowego interpretowanego języka **python** w wersji 3.8. Python jest dostępny na większości współczesnych platform oraz jest to język open-source. Dodatkowo wykorzystałem moduł **pygame**.

Pygame dodaje funkcjonalność do biblioteki SDL. Simple DirectMedia Layer(SDL) jest to wieloplatformowa biblioteka zapewniająca nisko poziomowy dostęp do audio, urządzeń peryferyjnych takich jak mysz, klawiatura, joystick oraz sprzętu graficznego (za pomocą OpenGL i Direct3D). Technologia open source zaimplementowana w C. Używana w grach firm takich jak *Valve Software* czy *SuperTuxCart*[3]. To za pomocą tej biblioteki pygame może odczytywać wejście czy rysować wyjście na ekranie.

Pygame pozwala tworzyć w pełni funkcjonalne gry oraz programy multimedialne w pythonie. Zaletami pygame są między innymi : podstawowe funkcje używają zoptymalizowanego kodu w C oraz Assembly, dostępny podobnie jak python na większości współczesnych platform. Ponadto jest modularny co pozwala programiście używać tylko tych komponentów, których naprawdę potrzebuje. Gier stworzonych z użyciem pygame na samej stronie projektu jest ponad 660 [4].

1.3 Implementacja

Głównym elementem implementacji jest nieskończona pętla, w której sprawdzam zdarzenia, które zaszły, sprawdzam czy mamy wejście od gracza, i aktualizuję obiekty gry oraz ekran. Na rysunku 1.2 opisana jako *Main game loop*.



Rysunek 1.2: Uproszczony diagram przepływu gry

Program zaczyna od wczytania wczytania zasobów gry – obrazków reprezentujących tło, rurę, ziemię, ptaka. Każdemu z tych zasobów tworzy obiekt klasy *pygame.image*, wczytywane są za pomocą metody *load*. Do śledzenia czasu w grze wykorzystywany jest obiekt klasy *pygame.time.Clock*.

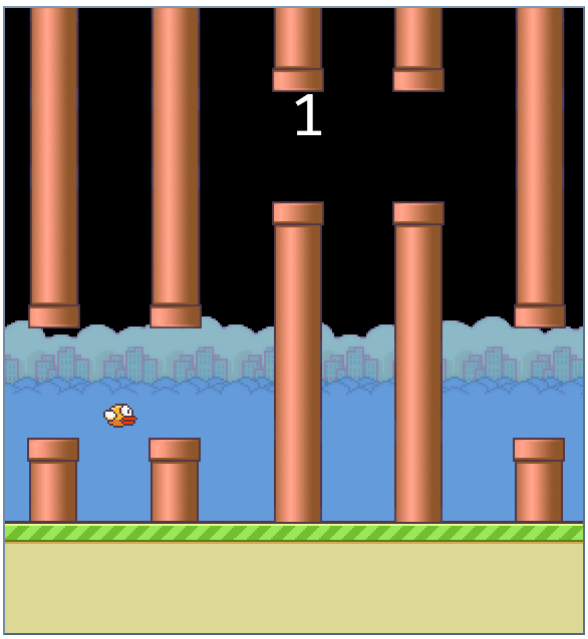
Stałe deklarowane na początku programu to na przykład: ilość jednostek o ile przesunąć w lewo obiekty rur przy każdym tyknięciu zegara, szerokość szczeliny między obiektami rur, czy czas po jakim generować nowe rury. W tym miejscu przypisujemy wartość stałej opisującej co jaki czas mamy generować nowe obiekty rur. Jest wyliczana ze wzoru:

$$P_f = 900 * (30 \div F) \tag{1.1}$$

Gdzie P_f to czas po jakim generowane są nowe obiekty rur (ang. *pipe frequency*), F to ilość klatek na sekundę (ang. *frames per second*). Potrzeba zmiennej częstotliwości generowania obiektów rur wzięła się z okoliczności, które opisuję w rozdziale Implementacja Q-learningu.

Przy uruchamianiu programu użytkownik ma opcję wybrać wartość parametru klatek na sekundę : 30, 60 bądź 120. Domyślnie jest to 30, stąd we wzorze znajduje się ta liczba. W ten sposób przy 30 klatkach na sekundę nowe obiekty rur będą generowane co 900 milisekund, przy 60 co 450 milisekund i co 225 milisekund w przypadku 120 klatek na sekundę. Po kilku próbach stwierdziłem, że częstsze generowanie obiektów rur sprawiało, że gra była zbyt trudna dla gracza lub w najgorszym wypadku sprawiało, że gra była nie do przejścia jak na przykład w scenariuszu przedstawionym na rysunku 1.3 gdzie obiekt ptaka nie będzie miał wystarczająco czasu by wznieść się na odpowiednią wysokość a potem opaść by ominąć rury. Częstsze generowanie obiektów rur nie ma wpływu przy większej ilości klatek na sekundę, gdyż te przemieszczają się w lewo co tyknięcie zegara o stałą ilość jednostek, a tykanie zegara uzależnione jest od ilości klatek na sekundę – większa ilość klatek na sekundę oznacza częstsze tykanie zegara przez co rury przemieszczają się z większą prędkością. Rozgrywka staje się zdecydowanie dynamiczniejsza i przez to również trudniejsza, ale tylko dla ludzi. Q-learningowy agent dostaje informacje o grze co tyknięcie zegara i jeszcze w tym samym tyknięciu podejmuje decyzje o ruchu, ale więcej o tym w rozdziale Implementacja Q-learningu

Po określeniu wartości stałych tworzymy trzy instancje klasy *pygame.sprite.Group*, jedną dla obiektu ptaka, drugą dla obiektów rur dolnych oraz jedną dla obiektów rur górnych. Będą nam potrzebne później przy sprawdzaniu czy zaszła kolizja między obiektem ptaka a obiektami rur. Sam obiekt ptaka jest instancją klasy *Bird* dziedziczącej po *pygame.sprite.Sprite*. Wspomniana klasa jest przeznaczona dla obiektów, które mają być widoczne na ekranie, zawiera między innymi pole na obrazek reprezentujący obiekt oraz metodę *update*[5], którą będę wywoływał co każdy obrót głównej pętli programowej. W przypadku obiektu ptaka w przeładowanej metodzie *update* znajduje się logika “latania”



Rysunek 1.3: Często generowane obiekty rur

– co wywołanie metody update inkrementuję zmienną która mówi o ile ma być przesunięty obiekt ptaka w dół do maksymalnej wartości ośmiu jednostek. Jeżeli został wciśnięty lewy przycisk myszy to wartość ta jest zmniejszana o dziesięć (minusowe wartości tej zmiennej powodują, że obiekt zostanie przesunięty w górę).

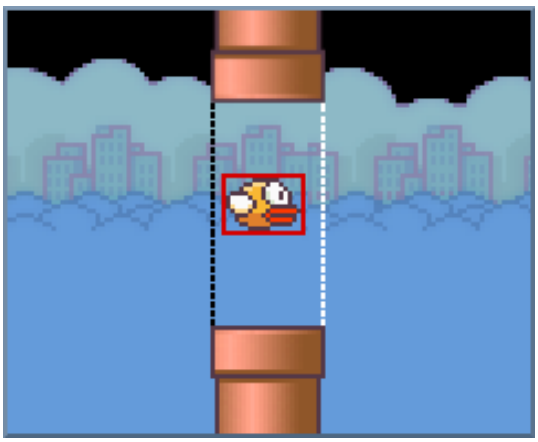
Po tym zaczyna się główna pętla programowa. Co P_f czasu generujemy nowe rury – polega to na stworzeniu dwóch instancji klasy *Pipe* dziedziczącej po *pygame.sprite.Sprite*. Generuję liczbę losową r z zakresu $(-100, 100)$, która będzie potrzebna w ustaleniu wysokości, na której wygeneruję obiekty rur. W konstruktorze nadaję obiektom odpowiedni obrazek. Po wczytaniu obrazku używam *get_rect* będącej metodą klasy *pygame.Surface*, której instancją jest nadany w konstruktorze obrazek. Zwrócony obiekt przypisuję do pola *rect*. Metoda ta mapuje obrazek na najmniejszy prostokąt, w którym zmieści się wczytany obrazek. Mamy dzięki temu dostęp do jego krawędzi, którym możemy nadawać współrzędne na ekranie, w których ma zostać narysowany. Wszystkie obiekty rysowane są w przestrzeni kartezjańskiej, gdzie punkt $(0, 0)$ jest lewym górnym rogiem tła. Górnej lewej krawędzi obiektu dolnej rury nadajemy współrzędne:

$$(W, H \div 2 + r + G \div 2)$$

Natomiast dolnej lewej krawędzi obiektu górnej rury nadajemy współrzędne:

$$(W, H \div 2 + r - G \div 2)$$

Gdzie W to szerokość tła ustalona na początku, H to wysokość tła, a G to stała oznaczająca odległość między dolną a górną rurą, przypisywana na początku programu. Tak



Rysunek 1.4: Granice *rect* obiektów *sprite*

narysowane obiekty rur zostaną narysowane zaraz za ekranem z prawej strony, przesunięte od środku ekranu x oraz połowę odległości ustalonej na początku działania programu. Tak wygenerowane obiekty dodaję do odpowiednich instancji *pygame.sprite.Group*. Jako, że klasa *Pipe* dziedziczy po *pygame.sprite.Sprite* (tak samo jak klasa *Bird*), to powinna przeładowywać metodę *update*[5]. W przypadku klasy reprezentującej rury jest tam tylko logika przesuwania obiektu w lewo co sprowadza się do zmniejszenia współrzędnej x zmiennej *rect* oraz ewentualnym usunięciu go jeżeli współrzędna ta po zmniejszeniu staje się ujemna (oznacza to, że jest po za lewą krawędzią ekranu przez co jest nie widoczna i nie potrzebna). Usuwanie polega na wywołaniu metody *kill*, co powoduje, że obiekt zostaje usunięty z każdej grupy, której był elementem.

W następnym kroku rysuję na ekranie obiekt ptaka za pomocą metody *draw* wywoływanej na grupie, do której ten obiekt należy. Metoda ta korzysta z nadanego w konstruktorze obrazka, oraz zmapowanego obiektu prostokąta (*sprite.Rect*) do określenia pozycji, na której ma zostać narysowany[6]. Rysowanie obiektów rur odbywa się w ten sam sposób, z tym, że wspomniana metoda jest wywoływana na odpowiednio grupie obiektów rur dolnych i górnych. Następnie na grupie, do której należy obiekt ptaka wywołuję metodę *update*, która obsługuje logikę "latania".

Najbliższym z prawej strony obiektem rury jest zawsze obiekt o indeksie zero z grupy dolnych bądź górnych rur, jest tak dzięki logice metody *update* klasy *Pipe*. Aby sprawdzić, czy przyznać graczowi punkt sprawdzamy najpierw, czy obiekt ptaka znalazł się w strefie między dwoma rurami, w tym celu wystarczy sprawdzić czy współrzędna x pola *rect.left* obiektu ptaka(na rysunku 1.4 zwizualizowana jako lewa krawędź czerwonego prostokąta wokół obiektu ptaka) jest większa tej samej współrzędnej obiektu dolnej rury. Na rysunku 1.4 jest to zaznaczone czarną przerywaną linią z lewej strony. Jeżeli tak jest to ustawiam logiczną flagę reprezentującą to zajście. W następnych obrotach głównej pętli programowej sprawdzane jest, czy ta flaga jest ustawiona i jeżeli tak jest to sprawdzam,

czy ptak opuścił strefę między dwoma rurami. Dzieje się to analogicznie do sprawdzania, czy wszedł z tą różnicą, że porównuję x pola *rect.right* obiektu dolnej rury (to znaczy współrzędną prawej krawędzi dolnej rury, na rysunku 1.4 oznaczona białą przerywaną linią z prawej strony) z współrzędną x pola *rect.left* obiektu ptaka. Jeżeli tak się stało to punkt zostaje przyznany a wcześniej wspomnianą flaga zresetowana. Powody takiego rozwiązania są dwa:

- Jeżeli sprawdzałbym tylko czy ptak ominął tylko lewą, albo tylko prawą cześć pola *rect* obiektu dolnej rury to punkty byłyby naliczane przy każdym obrocie głównej pętli programowej gdy obiekt rury najbliższej z prawej został ominięty a nie dotarł jeszcze za lewą granicę ekranu i nie został jeszcze usunięty.
- Jeżeli przyznawałbym punkt, gdy tylko obiekt ptaka znajdzie się w strefie między dwoma obiektami rurami to byłaby szansa, że gracz wpadnie na obiekt rury zaraz po przyznaniu punktu – byłoby to nieprawidłowe zachowanie

Aby sprawdzić, czy zaszła kolizja obiektu ptaka z ziemią wystarczy sprawdzić, czy współrzędna y dolnej części *rect* (czyli *rect.bottom*) będąca zmienną obiektu ptaka jest większa od H (przypomnę, że w *pygame* układ współrzędnych kartezjańskich ma swój początek w lewym górnym rogu ekranu a współrzędne z osi OY "rosną w dół"), gdzie H to wysokość tła przypisana na początku. Sprawdzenie kolizji między obiektem ptaka a obiektami rur dzięki *pygame* sprowadza się do wywołania metody **pygame.sprite.groupcollide**, która sprawdza, czy doszło do kolizji pomiędzy dwiema grupami podanymi jako argumenty i zwraca słownik, gdzie kluczami są obiekty z grupy podanej jako pierwszy argument a wartościami każdy obiekt z grupy podanej jako drugi argument, z którym obiekt będący kluczem wszedł w kolizję. Pod spodem metoda ta sprawdza tak naprawdę czy zmienne *rect* obiektu z każdej grupy nie nachodzą na siebie[7]. Jako, że grupa obiektów *Bird* ma tylko jeden element, to wystarczy, że wspomniana metoda zwróci cokolwiek by stwierdzić, że zaszła kolizja między obiektem ptaka a obiektami rur i zakończyć grę.

Pygame.event to moduł, który pozwala na interakcję z kolejką wydarzeń, które odbywają się w systemie np. wciśnięto lub zwolniono przycisk przycisk myszy bądź klawiatury, zmieniono rozdzielczość ekranu, wyłączono program. To ten moduł komunikuje się z biblioteką SDL. Jest zależny od modułu **pygame.display**[8], który zarządza wyświetlaniem okna programu na ekranie. Okno programu jest inicjalizowane na początku działania programu na podstawie wymiarów obrazków reprezentujących tło i ziemię w grze. Sprawdzanie wydarzeń polega na iteracji kolejki, którą *pygame* udostępnia za pomocą metody *pygame.event.get*. Jeżeli zajdzie się w niej wydarzenie o typie *pygame.QUIT* albo *pygame.KEYDOWN* (został wciśnięty przycisk klawiatury) oraz *pygame.event.key* to *pygame.K_ESCAPE* (wciśnięty klawisz to escape) to podobnie jak w przypadku wykrycia kolizji gra jest zakańczana.

Dodatkowo *pygame* umożliwia komunikację ze sprzętem bezpośrednio, nie przez

sprawdzanie kolejki wydarzeń. Mysz obsługiwana jest przez moduł **pygame.mouse**. Pozwala na sprawdzenie w każdym momencie, które przyciski na myszy są wciśnięte za pomocą metody *pygame.mouse.get_pressed*, która zwraca tablicę wartości logicznych reprezentujących przyciski na myszy. Sprawdzenie, czy gracz wcisnął lewy przycisk myszy jest sprawdzane w metodzie *update* obiektu ptaka i polega na sprawdzeniu dwóch warunków:

- Czy lewy przycisk myszy jest wciśnięty
- Czy lewy przycisk myszy był wciśnięty w ostatnim obrocie głównej pętli programowej

Jeżeli przycisk nie był wciśnięty to obiekt ptaka podlatuje i ustawiana jest flaga logiczna mówiąca o tym, że lewy przycisk myszy jest wciśnięty. Flaga ta jest resetowana, gdy przy następnym obrocie głównej pętli programowej przycisk nie będzie wciśnięty. Takie sprawdzanie jest konieczne, inaczej jedno wciśnięcie lewego przycisku myszy powodowałoby, że obiekt ptaka podlatywał by do góry o dużą ilość jednostek, dopóki przycisk nie zostałby zwolniony.

Na końcu aktualizowane jest okno gry dzięki użyciu metody **pygame.display.update**, która wysyła zaktualizowany obraz na ekran. Optymalizacja jaka jest zastosowana w tej metodzie, sprawia, że aktualizowane są tylko te części obrazu, które faktycznie się zmieniły a nie całość.

Gdy gra zostaje zakończona, na ekranie rysowany przycisk reset oraz ustawiana flaga logiczna zatrzymująca wykonywanie opisanych w tym rozdziale dyrektyw dopóki nie zostanie wciśnięty lewy przycisk myszy, wtedy gra jest resetowana, to znaczy wszystkie zmienne są ustawiane do stanu początkowego co powoduje rozpoczęcie gry na nowo.

Rozdział 2

Uczenie maszynowe

2.1 Czym jest sztuczna inteligencja

Sztuczna inteligencja (ang. *artificial intelligence*) to dziedzina nauki, która zajmuje się syntezowaniem oraz analizowaniem obliczeniowych agentów, którzy potrafią podejmować inteligentne decyzje.

Agent jest czymś, co potrafi podejmować decyzje w pewnym środowisku, robić coś. Agentami są na przykład: psy, termostaty, ludzie, korporacje, samoloty czy roboty.

Mówimy, że agent działa/podejmuje decyzje inteligentnie gdy:

- Uczy się na podstawie swojego doświadczenia
- Jest elastyczny na zmieniające się środowisko i zmieniające się cele
- Decyzje, które podejmuje są właściwe dla okoliczności, w których się znajduje oraz celów, które ma osiągnąć, biorące pod uwagę krótko i długo terminowe konsekwencje

Obliczeniowy agent to taki agent, którego podejmowane decyzje mogą zostać wyjaśnione pod względem obliczeniowym. To znaczy decyzja może zostać rozbita na podstawowe operacje czyli takie, które mogą zostać zaimplementowane na fizycznym urządzeniu. Takie obliczenia mogą przyjmować wiele różnych form. U ludzi są one wykonywane w mózgu, w przypadku komputerów na procesorach.

Wszyscy agenci mają swoje ograniczenia. Żaden agent nie jest wszechmocny czy wszechwiedzący. Agenci mogą jedynie obserwować swoje ograniczone środowisko w bardzo wyspecjalizowanych domenach. Agenci mają skończoną pamięć oraz nie mają nieskończonej ilości czasu na podjęcie decyzji.

Naukowym celem sztucznej inteligencji jest zrozumienie zasad, które sprawiają, że inteligentne zachowania są możliwe w naturalnych czy sztucznych systemach. Jest on osiągany przez:

- Analizę naturalnych i sztucznych agentów

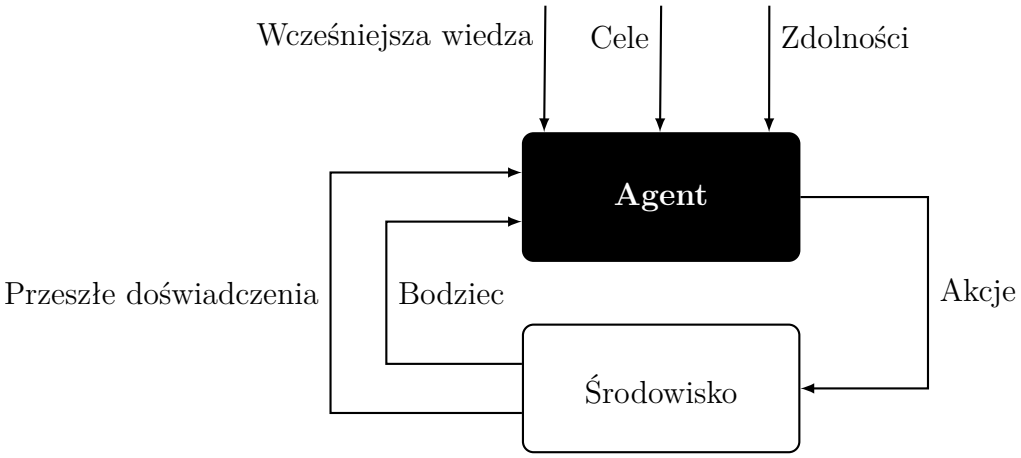
- Formułowaniem i sprawdzaniem hipotez dotyczących czego potrzeba by skonstruować inteligentnych agentów
- Projektowanie, budowanie i eksperymentowanie z systemami obliczeniowymi, które wykonują zadania powszechnie uważane za wymagające inteligencji

Inżynieryjnym celem sztucznej inteligencji jest projektowanie i syntezowanie użytecznych inteligentnych artefaktów. Chcemy tworzyć agentów, którzy będą działać inteligentnie. Tacy agenci są użyteczni w wielu problemach.

2.2 Interakcja agenta ze środowiskiem

Kwintesencją sztucznej inteligencji jest praktyczne rozumowanie, czyli rozumowanie w celu osiągnięcia jakiegoś celu. Na agenta składa się zespojenie percepcji, rozumowania oraz podejmowania działań. Agent podejmuje działania w **środowisku**.

Agentem może być na przykład komputer z sensorami i siłownikami, innymi słowy robot, gdzie jego środowiskiem jest świat rzeczywisty. Może być to samo prowadzący się samochód, lub jak w moim przypadku może być to program, który działa w czysto obliczeniowym środowisku – agent programowy (ang. *software agent*).



Rysunek 2.1: Schemat interakcji agenta ze środowiskiem

Rysunek 2.1 pokazuje widok czarnej skrzynki agenta z wejściami i wyjściami. W dowolnym momencie agent jest zależny od:

- **Wcześniejszej wiedzy** o agencie i środowisku
- **Bodźców** otrzymanych od obecnego środowiska, które mogą zawierać obserwacje o środowisku, ale również akcje, które środowisko narzuca agentowi
- **Przeszłych doświadczeń** z poprzednich działań i bodźców lub innych danych, z których może się uczyć

- **Celów**, które musi starać się osiągnąć
- **Zdolności**, czyli prymitywne działania, które agent może wykonać

W czarnej skrzynce, agent jest w pewnym stanie wiedzy, może być to wiedza o środowisku, o tym co próbuje osiągnąć i co zamierza zrobić. Agent aktualizuje ten stan bazując na bodźcach ze środowiska. Na podstawie stanu wiedzy decyduje o swoich działaniach[9].

2.3 Wiedza agenta

W najprostszym przypadku, kiedy agent decyduje co powinien zrobić odnosi się do modelu środowiska opartego o stany, z celem do osiągnięcia i bez żadnych niepewności. Agent jest w stanie ustalić jak osiągnąć swój cel przeszukując przestrzeń stanów środowiska, tak by ze swojego obecnego stanu przejść do stanu, w którym cel jest osiągnięty. Mając kompletną przestrzeń stanów, próbuje znaleźć sekwencję działań, których podjęcie sprawi, że osiągnie cel, zanim jakiekolwiek działania podejmie.

Przestrzeń stanów (ang. *state space*) to jedno z ogólnych sformułowań inteligentnych działań. **Stan** zawiera wszystkie informacje potrzebne do przewidzenia efektów danego działania i ustalenia, czy tenże stan jest potrzebny do osiągnięcia celu. Przeszukiwanie przestrzeni stanów zakłada:

- Agent ma całkowitą wiedzę o przestrzeni stanów i planuje na przypadek, w którym obserwuje, w którym stanie się znajduje – jest całkowita obserwowalność
- Agent ma zbiór akcji, które mają znane, deterministyczne efekty na środowisko
- Agent może ustalić czy stan spełnia cel

Rozwiązanie to sekwencja działań, które przeniosą agenta z jego obecnego stanu do stanu, który spełnia cel.

2.4 Proces decyzyjny

W poprzednim podrozdziale założyłem, że rozwiązanie to skończona sekwencja działań, co jednak, jeżeli problem, który agent, musi rozważyć trwający proces lub nie wie ile działań będzie musiał jeszcze podjąć. Takie problemy nazywane są **problemami nieskończonego horyzontu** (ang. *infinite horizon problems*), gdzie proces może trwać wiecznie, jak na przykład problem grania we “Flappy Bird” lub **problemami nieokreślonego horyzontu** (ang. *indefinite horizon problems*), w którym agent w końcu zakończy podejmowanie działań, ale nie wie kiedy to nastąpi.

Dla trwających procesów, może nie mieć sensu rozważanie użyteczności na końcu procesu ponieważ agent może nigdy nie dojść do końca. Zamiast tego agent może otrzymać sekwencję **nagród** (ang. *rewards*). Te nagrody włączają koszt działania, oprócz nagród

mogą agentowi mogą być nadane również kary. Negatywne nagrody nazywane są **karami** (ang. *punishments*). Problemy niezdefiniowanego horyzontu mogą być wymodelowane używając stanu stop. **Stan stop** bądź **stan absorbujący** to stan, w którym wszystkie działania nie mają żadnego efektu to znaczy jeżeli agent znajdzie się w tym stanie to wszystkie jego działania wracają do stanu z zerową nagrodą. Osiągniecie celu może być wymodelowane w ten sposób, by agent otrzymał nagrodę właśnie, kiedy osiągnie stan stopu.

2.4.1 Łańcuchy Markova

Mówimy, że zmienna X jest **warunkowo niezależna** od losowej zmiennej Y zakładając, że mamy zbiór losowych zmiennych Z_s jeżeli

$$P(X|Y, Z_s) = P(X|Z_s)$$

gdzie prawdopodobieństwa są dobrze zdefiniowane (żadne nie jest równe 0). Zapis $P(X|Y, Z_s)$ oznacza prawdopodobieństwo zajścia X pod warunkiem, że zajdzie Y oraz Z_s. To oznacza, że dla każdego $x \in$ dziedzina(X), dla każdego $y \in$ dziedzina(Y) oraz dla każdego $z \in$ dziedzina(Z_s) jeżeli $P(Y = y \wedge Z_s = z) > 0$,

$$P(X = x|Y = y \wedge Z_s = z) = P(X = x|Z_s = z).$$

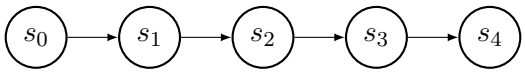
To znaczy, mając wartość każdej zmiennej Z_s, wiedza o prawdopodobieństwie Y nie ma wpływu na przekonanie o X.

Pojęcie warunkowej niezależności jest użyte by nadać zwięzłą reprezentację wielu domenom. Chodzi o to, że jeżeli mamy daną losową zmienną X, to może istnieć kilka zmiennych, które bezpośrednio wpływają na wartość X, w pewnym sensie X jest warunkowo niezależne od innych zmiennych biorąc pod uwagę te zmienne. Zbiór lokalnie wpływających na X zmiennych jest nazwany **ogrodzeniem Markova** (ang. *Markov blanket*). Ta lokalność jest wykorzystywana w sieciach przekonań[10].

Sieć przekonań (ang. *belief network*), również nazywana **siecią Bayesa** to acykliczny graf skierowany, gdzie wierzchołki są losowymi zmiennymi. Ponadto istnieje krawędź od każdego elementu ze zbioru rodzice(X_i) do X_i. Powiązany z siecią przekonań jest zbiór rozkładu prawdopodobieństw warunkowych, które określają warunkowe prawdopodobieństwo między każdą zmienną a jej rodzicami (co włącza wcześniejsze prawdopodobieństwa zmiennych bez rodziców). A więc sieć przekonań składa się z:

- Grafu acyklicznego skierowanego, gdzie każdy wierzchołek jest oznaczony losową zmienną
- Dziedziny dla każdej losowej zmiennej
- Zbioru rozkładu prawdopodobieństw warunkowych przypisujący $P(X|\text{rodzice}(X))$ dla każdej zmiennej X

Łańcuch Markova to sieć przekonań z losowymi zmiennymi w sekwencji, gdzie każda zmienna bezpośrednio zależy od swojego poprzednika w sekwencji. Łańcuchy Markova są używane do reprezentowania sekwencji wartości jak na przykład sekwencja stanów w dynamicznym systemie lub sekwencja słów w zdaniu. Każdy punkt w sekwencji jest nazywany **etapem** (ang. *stage*). Rysunek 2.2 pokazuje ogólny łańcuch Markova jako sieć



Rysunek 2.2: Łańcuch Markova jako sieć przekonań

przekonań. Sieć ma pięć etapów, ale nie musi zatrzymywać się na s_4, może rozrastać się w nieskończoność. Sieci przekonań przekazują założenie niezależności:

$$P(S_{i+1}|S_0, \ldots, S_i) = P(S_{i+1}|S_i),$$

które jest nazywane **założeniem Markova**

Często sekwencje są rozłożone w czasie, wtedy S_t reprezentuje stan w momencie t. Intuicyjnie S_t przekazuje całą informację o historii, która mogłaby wpłynąć na przyszłe stany. Niezależność przypuszczenia w łańcuchu Markova może być rozumiana jako "biorąc pod uwagę teraźniejszość – przyszłość jest warunkowo niezależna od przeszłości"

Mówimy, że łańcuch Markova jest **modelem stacjonarnym** (ang. *stationary model*) oraz **czasowo homogenicznym modelem** (ang. *time-homogenous model*) jeżeli wszystkie zmienne mają tą samą dziedzinę oraz wszystkie prawdopodobieństwa przejść są takie same dla każdego etapu, to znaczy:

$$\forall i \geq 0, P(S_{i+1}|S_i) = P(S_1|S_0)$$

By określić stacjonarny łańcuch Markova, podano dwa prawdopodobieństwa warunkowe:

- $P(S_0)$ specyfikuje wszystkie początkowe warunki
- $P(S_{i+1}|S_i)$ specyfikuje **dynamikę**, która jest taka sama dla każdego $i \geq 0$

Stacjonarne łańcuchy Markova interesują nas z kilku względów:

- Zapewniają prosty model, który jest łatwy do określenia
- Założenie stacjonarności jest często naturalnym modelem, ponieważ dynamika środowiska zazwyczaj nie zmienia się w czasie.
- Sieć może się rozrastać w nieskończoność. Określenie małej liczby parametrów daje nam nieskończoną sieć[11].

2.4.2 Sieć decyzyjna

Zazwyczaj agent nie podejmuje decyzji bez wcześniejszych obserwacji środowiska, nie podejmuje też tylko jednej decyzji. Bardziej typowy scenariusz wygląda następująco – agent obserwuje swoje środowisko, decyduje jaką akcję podjąć, wykonuje tą akcję, obserwuje zmienione po jego akcji środowisko, i tak dalej. Następujące po sobie (sekwencyjne) akcje agenta mogą zależeć od tego co agent zaobserwuje, a to co zaobserwuje może zależeć na poprzednich podjętych przez niego akcji. W takim wypadku, często jedynym powodem dla wykonania jakiejś akcji jest to by dostarczyć kontekst dla przyszłych akcji. Akcje wykonywane tylko po to by nabyć informacje nazywane są **akcjami poszukującymi informacji**. Formalnie nie muszę odróżniać akcji poszukujących informacji od innych akcji. Zazwyczaj podjęte akcje będą prowadziły do pozyskania nowych informacji i wywarciu jakiejś zmiany na środowisku.

Sekwencyjny model decyzyjny modeluje:

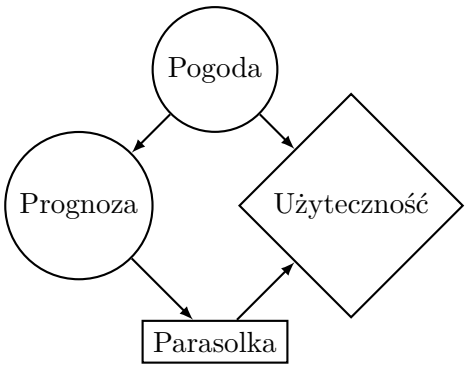
- Które akcje są dostępne dla agenta, w którym etapie
- Jaka informacja jest, albo będzie dostępna dla agenta, kiedy będzie musiał podjąć akcję
- Efekty akcji na środowisko
- To jak pożądane te efekty będą

Sieć decyzyjna zwana również **diagramem wpływu** to graficzna reprezentacja skończonego problemu decyzji sekwencyjnych[12]. Sieć decyzyjna rozszerza sieć przekonań by zawierać zmienne decyzji (akcji) i użyteczności.

W szczególności, sieć decyzyjna to skierowany graf acykliczny, z następującymi wierzchołkami:

- **Wierzchołek decyzji**, rysowany jako prostokąt reprezentuje zmienne decyzji. Agent może wybrać wartość każdej ze zmiennych decyzji.
- **Wierzchołek szansy**, rysowany jako okrąg reprezentuje zmienne losowe. To są te same wierzchołki co w sieciach przekonań. Każdy wierzchołek szansy ma powiązaną ze sobą domenę i warunkowe prawdopodobieństwo biorące pod uwagę rodziców wierzchołka. Tak jak w sieciach przekonań, rodzice wierzchołka reprezentują warunkową zależność: zmienna jest niezależna od innych zmiennych nie będących jej potomkami. W sieci decyzyjnej zarówno wierzchołki decyzji i wierzchołki szansy mogą być rodzicami wierzchołka szansy.
- **Wierzchołek użyteczności**, rysowany jako romb reprezentuje użyteczność. Zarówno wierzchołki szansy jak i wierzchołki decyzji mogą być rodzicami wierzchołka użyteczności.

Krawędzie wchodzące do wierzchołków decyzyjnych reprezentują informację, która będzie dostępna, jeżeli decyzja zostanie podjęta. Krawędzie wchodzące do wierzchołków



Rysunek 2.3: Sieć decyzyjna reprezentująca decyzję, czy wziąć parasolkę

szansy reprezentują zależność probabilistyczną. Krawędzie wchodzące do wierzchołków użyteczności reprezentują od czego ta użyteczność zależy.

Mówimy, że **agent nie zapomina**, jeżeli decyzje, które podjął są uporządkowane w czasie, i agent pamięta swoje poprzednie decyzje i każdą informację, która była dostępna przy poprzedniej decyzji.

Mówimy, że **sieć decyzyjna nie zapomina**, jeżeli jej wierzchołki decyzji są całkowicie uporządkowane, to znaczy, jeżeli wierzchołek decyzyjny D_i jest przed D_j to znaczy, że D_i jest rodzicem D_j i każdy rodzic D_i jest również rodzicem D_j. A to oznacza, że każda informacja dostępna dla D_i jest dostępna dla każdej późniejszej decyzji w sekwencji i akcja wybrana dla decyzji D_i jest częścią informacji dostępnej dla każdej późniejszej decyzji.

2.4.3 Proces decyzyjny Markova

Proces decyzyjny Markova, może być rozumiany jako łańcuch Markova powiększony o działania agenta oraz nagrodę dla agenta. W każdym etapie agent decyduje jakie działanie podjąć, nagroda i nowy stan zależą od podjętego przez agenta działania oraz poprzedniego stanu.

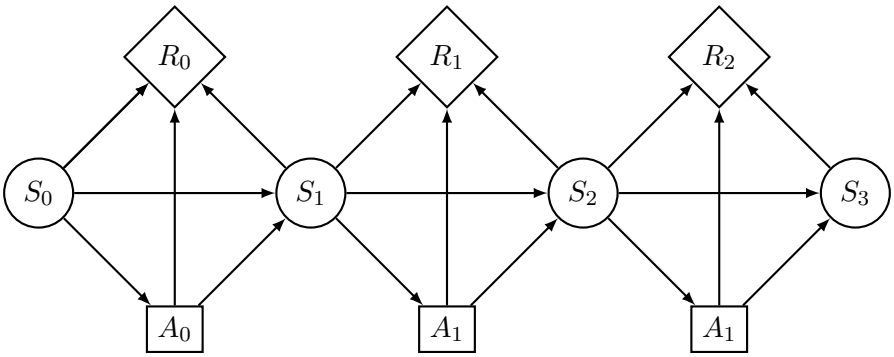
Rozważę jedynie stacjonarne modele łańcuchu Markova, gdzie przejścia między stanami oraz nagrody nie zależą od czasu.

Proces decyzyjny Markova (ang. *Markov decision process (MDP)*) składa się z:

- S – zbiór stanów środowiska
- A – zbiór decyzji (akcji możliwych do podjęcia przez agenta)
- $P : S \times S \times A \to [0, 1]$ – funkcja, która określa **dynamikę**. Zapis $P(s'|s, a)$ oznacza prawdopodobieństwo, że agent przejdzie do stanu s' pod warunkiem, że był w stanie s i podjął akcję a, czyli:

$$\forall s \in S \ \forall a \in A \sum_{s' \in S} P(s'|s, a) = 1$$

- $R : S \times S \times A \times S \to \mathfrak{R}$, gdzie $R(s, a, s')$ to **funkcja nagrody**, dająca oczekiwaną natychmiastową nagrodę za wykonanie działania a i przejścia do stanu s' ze stanu s. Czasem wygodnie jest użyć zapisu $R(s, a)$ – oczekiwana wartość wykonania a w stanie s, która równa jest $R(s, a) = \sum_{s'} R(s, a, s') * P(s'|s, a)$.



Rysunek 2.4: Sieć decyzyjna reprezentująca skończoną część MDP

2.5 Q-learning

Tutaj wszystko związane z qlearningiem, wzory, itp :)

Rozdział 3

Implementacja Q-learningu

Tutaj fragmenty kodu, jakie zmiany dokonałem względem książkowego algorytmu

3.1 Podejście pierwsze grid 10 na 10

Po kolei wyjaśnienie co to grid 10 na 10 jak to jest implementowane itp

3.2 Podejście drugie grid 10 na 10 i flaga za śmierć od wysokiej rury

Jak wyżej

3.3 Podejście trzecie grid 5 na 5 z flagą

Jak wyżej

Rozdział 4

Podsumowanie

Wyniki osiągane są w miarę ok ale mogło by być lepiej bo są na internecie lepsze podejścia do tego problemu – być może wynika to z tego ze sam implementowałem grę i nie jest ona tak dobrze przetestowana jak ta dostępna na GitHub?

4.1 Co można by ulepszyć

Tutaj będę pisał co mogłem zrobić gdybym poświęcił więcej czasu np:

- uczenie bez wizualizacji pygamowej
- zrównoleglenie by paru agentów na raz mogło się uczyć
- doszlifowanie algorytmu??

Bibliografia

[1] K. Dabérius, E. Granat, and P. Karlsson, "Deep execution - value and policy based reinforcement learning for trading and beating market benchmarks," 2019. [Online]. Available: http://dx.doi.org/10.2139/ssrn.3374766

[2] "Strona internetowa dotgears." [Online]. Available: https://dotgears.com/games/flappybird

[3] "Strona internetowa sdl." [Online]. Available: https://www.libsdl.org/

[4] "Strona about projektu pygame." [Online]. Available: https://www.pygame.org/wiki/about

[5] *Dokumentacja klasy pygame.sprite.Sprite*. [Online]. Available: https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite

[6] *Dokumentacja metody pygame.sprite.Group.Draw*. [Online]. Available: https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group.draw

[7] *Dokumentacja metody pygame.groupcollide*. [Online]. Available: https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.groupcollide

[8] *Dokumentacja modułu pygame.event*. [Online]. Available: https://www.pygame.org/docs/ref/event.html

[9] D. Poole and A. Mackworth, *Artificial Intelligence Foundations of Computational Agents*, 2nd ed. Cambridge University Press, 2017, ch. Agents Situated in Environments.

[10] ——, *Artificial Intelligence Foundations of Computational Agents*, 2nd ed. Cambridge University Press, 2017, ch. Belief Networks.

[11] ——, *Artificial Intelligence Foundations of Computational Agents*, 2nd ed. Cambridge University Press, 2017, ch. Markov Chains.

[12] ——, *Artificial Intelligence Foundations of Computational Agents*, 2nd ed. Cambridge University Press, 2017, ch. Decision Networks.