AGH
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Mateusz Winiarski

kierunek studiów: **informatyka stosowana**

Sztuczna inteligencja w grach komputerowych na przykładzie logiki rozmytej, algorytmu stada i problemu najkrótszej ścieżki w grze 2D

Opiekun: **dr inż. Janusz Malinowski**

Kraków, styczeń 2017

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

……

(czytelny podpis)

Ocena merytoryczna opiekuna.

Ocena merytoryczna recenzenta.

Spis treści:

1. Wstęp 7

2. Algorytm szukania najkrótszej ścieżki 9

2.1. Graf w problemie najkrótszej ścieżki 9
2.1.1. Graf 9
2.1.2. Graf z nieujemnymi wagami 9
2.1.3. Graf skierowany 10
2.2. Algorytm Dijkstry 10
2.2.1. Problem 10
2.2.2. Rozwiązanie 11
2.2.2.1. Przetwarzanie obecnego węzła 11
2.2.2.2. Listy węzłów 11
2.2.2.3. Obliczanie całkowitego kosztu dla listy zamkniętych i otwartych 11
2.2.2.4. Zakończenie algorytmu 12
2.2.2.5. Rekonstrukcja ścieżki 12
2.2.3. Prezentacja działania 13
2.3. Algorytm A* 13
2.3.1. Problem 14
2.3.2. Rozwiązanie 14
2.3.2.1. Heurystyka 14
2.3.2.1.1. Heurystyka euklidesowa 15
2.3.2.1.2. Heurystyka oparta na metryce Manhattan 15
2.3.2.1.3. Heurystyka null 15
2.3.2.2. Zmiany w algorytmie wynikające z zastosowania heurystyki 15
2.3.3. Prezentacja działania 16
2.4. Implementacja 19
2.4.1. Funkcje heurystyczne 19
2.4.1.1. Heurystyka euklidesowa 19
2.4.1.2. Heurystyka Manhattan 19
2.4.1.3. Heurystyka null 19
2.4.2. Klasa reprezentująca węzeł grafu wraz z połączeniami 19
2.4.3. Funkcja rekonstruująca ścieżkę 20
2.4.4. Funkcja algorytmu A* 20
2.5. Testowanie 23
2.5.1. Testy wydajnościowe 25
2.5.2. Testy jakościowe 28

3. Algorytm stada 30

3.1. Problem 30
3.2. Rozwiązanie 30
3.2.1. Reguła rozdzielności 30
3.2.2. Reguła wyrównania 30
3.2.3. Reguła spójności 30
3.2.4. Reguła unikania 31
3.3. Implementacja 31
3.3.1. Podstawowe własności klasy Animal 31
3.3.2. Reguła rozdzielności 32

3.3.3. Reguła wyrównania ... 32
3.3.4. Reguła spójności ... 32
3.3.5. Zarządzanie stadem ... 33
4. Logika rozmyta ... 35
4.1. Podstawowa różnica względem logiki klasycznej ... 35
4.2. Zbiór rozmyty i rozmyta zmienna lingwistyczna ... 35
4.3. Funkcja przynależności do zbioru ... 36
4.4. Operacje na zbiorach rozmytych ... 37
4.5. Algorytm wnioskowania rozmytego ... 39
4.5.1. Rozmycie (fuzyfikacja) ... 39
4.5.2. Wnioskowanie rozmyte ... 40
4.5.3. Wyostrzenie (defuzyfikacja) ... 40
4.5.3.1. Najwyższy stopień przynależności ... 40
4.5.3.2. Łączenie w oparciu o funkcje przynależności ... 41
4.5.3.3. Środek ciężkości ... 41
4.6. Implementacja ... 42
4.6.1. Klasa zmiennej rozmytej ... 42
4.6.2. Funkcje przynależności do zbioru rozmytego ... 43
4.6.2.1. Funkcje left i right shoulder ... 44
4.6.2.2. Funkcja triangularna ... 44
4.6.2.3. Funkcja trapezoidalna ... 45
4.6.3. Zaimplementowane zmienne rozmyte ... 45
4.6.3.1. FuzzyHP ... 45
4.6.3.2. FuzzyLaziness ... 46
4.6.3.3. FuzzyRest ... 46
4.6.3.4. FuzzyMorale ... 47
4.6.4. Rozmyta macierz asocjacyjna ... 47
4.6.5. Implementacja reguł rozmytych ... 48
4.6.5.1. Przykład obliczeń reguł rozmytych ... 49
5. Opis wykorzystanych narzędzi ... 52
5.1. SFML.Net ... 52
5.1.1. SFML.System ... 52
5.1.2. SFML.Window ... 52
5.1.3. SFML.Graphic ... 52
5.1.4. SFML.Audio ... 53
6. Prezentacja gry ... 54
7. Podsumowanie ... 57
Bibliografia ... 58

1. Wstęp

Pierwsze prototypy gier komputerowych zostały stworzone w latach 50. i 60. XX wieku. Alexander Sandy Douglas w ramach swojej pracy doktorskiej z roku 1952 stworzył *Noughts and Crosses* bazującą na grze w kółko i krzyżyk. Później William Higinbotham w roku 1958 stworzył na bazie oscyloskopu symulację tenisa stołowego nazwaną *Tennis for Two*. Przełomową była gra *Spacewar!* stworzona w 1961 roku przez Steve'a Russella. Jest to pierwsza gra, którą bez wątpliwości można nazwać grą wideo, podczas gdy jej poprzednikom zarzuca się, że nie generują sygnału wideo[1]. Powyższe gry to produkcje akademickie, których grono odbiorców było niewielkie. Pierwszą grą, która odniosła sukces komercyjny, był *Pong*, wzorowany na *Tennis for Two*, a wydany w 1972 roku przez Atari[2].

Za pierwszą grę, w której agenci sterowani są za pomocą sztucznej inteligencji (ang. *artificial intelligence*, *AI*), uznaje się japońskiego *Pac-Mana* stworzonego w 1980 roku przez firmę Namaco[3]. Przeciwnikami są tam cztery duchy:

- czerwony – podąża za graczem jak cień, a jeśli Pac-Man zje odpowiednią liczbę kulek, przyspiesza;
- różowy – stara się przewidzieć ruch Pac-Mana i zaatakować go z zasadzki. W związku z tym często "współpracuje" z czerwonym duchem i zachodzi graczowi drogę, a ten drugi uniemożliwia mu ucieczkę;
- pomarańczowy – ma taktykę podobną do ducha czerwonego, lecz gdy znajdzie się za blisko Pac-Mana, zmienia zdanie i próbuje uciec do narożnika mapy. Gdy tylko Pac-Man wystarczająco się oddali, znowu rusza w pościg;
- błękitny – może przyjąć dowolną z powyższych taktyk. To, którą wybierze, zależy od pozycji lub kierunku patrzenia Pac-Mana, jak też od miejsca, w którym znajduje się duch czerwony. Jego decyzje mogą zmieniać się często i radykalnie.

Celem gier komputerowych jest przede wszystkim dostarczenie rozrywki, nie ma potrzeby modelowania pracy ludzkiego mózgu. Wystarczy więc, że zaprogramowana inteligencja postaci będzie w pewien sposób imitowała racjonalne działania i zachowania. W modelowaniu takich zachowań często wykorzystywane są sieci neuronowe, algorytmy genetyczne czy logika rozmyta.

Logika rozmyta w lepszy sposób odwzorowuje proces decyzyjny człowieka niż zwykłe użycie po sobie ciągu klauzuli `if ... else`. Wykorzystuje się ją często do symulacji pragnień i emocji. Dzięki niej akcje jednostki, bazujące na jej określonych cechach, potrafią być bardziej „racjonalne”. Daje to sporą różnorodność zachowań i nadaje przeciwnikom pewien indywidualizm.

Algorytm stada natomiast wykorzystywany jest do symulacji poruszania się dużych zbiorowisk obiektów. Dzięki kilku względnie prostym regułom pozwala nadać grupie realistyczne zachowania zbiorowe. Podobne są one do tych, jakie możemy obserwować w stadzie ptaków, ławicy ryb czy roju pszczół.

[1] *Historia gier komputerowych* [https://pl.wikipedia.org/wiki/Historia_gier_komputerowych]
[2] *Pong* [https://pl.wikipedia.org/wiki/Pong]
[3] *Pac-Man* [https://pl.wikipedia.org/wiki/Pac-Man]

Problem najkrótszej ścieżki powstaje wszędzie tam, gdzie sterowana komputerowo postać potrzebuje się przemieścić. W sporej części gier ścieżki te nie są stałe i trzeba wyznaczyć tę, po której obiekt ma się poruszyć.

Motywacją do napisania tej pracy była chęć poznania jednego z nowoczesnych i rozbudowanych języków programistycznych, jakim jest C# oraz moje zainteresowanie grami, w których buduje się pewne struktury i zarządza ludźmi. Są to takie gry jak *RimWorld*, gdzie gracz wciela się w rozbitków na obcej planecie i ma za cel wybudowanie kolonii, czy *Prison Architect*, w której gracz buduje więzienie, którym musi zarządzać i utrzymać więźniów w ryzach. Tak zrodził się pomysł na stworzenie własnej, prostej gry, w której gracz ma za zadanie zbudować osadę oraz bronić jej przed atakami przeciwników.

W pracy omówię algorytmy wyszukiwania najkrótszej ścieżki oraz pokażę ich porównanie wydajnościowe i jakościowe. Przedstawię także algorytm stada i omówię zastosowania logiki rozmytej do symulacji zachowań postaci.

2. ALGORYTM SZUKANIA NAJKRÓTSZEJ ŚCIEŻKI

W większości obecnych gier postacie poruszają się. O ile zaprogramowanie stałej trasy, na przykład patrolu, jest proste, gdyż można określić kilka punktów, między którymi postać będzie się przemieszczać, o tyle zaprogramowanie ruchu postaci tak, żeby umiała tę trasę znaleźć sama, nie jest już takie trywialne. Sprowadza się ono do znalezienia pewnej ścieżki, najlepiej jak najszybszej. Z angielskiego problem ten określa się mianem *pathfindingu*.

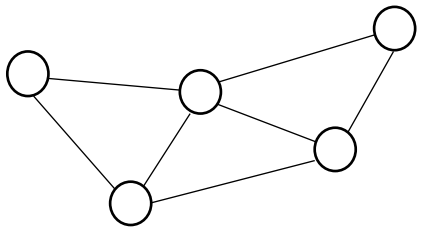
Większość gier do rozwiązania tego problemu wykorzystuje *algorytm Dijkstry* bądź też jego rozbudowaną, heurystyczną wersję – *algorytm A**. Głównym powodem ich wykorzystania jest fakt, że są one wydajne i łatwe do zaimplementowania[4].

2.1. GRAF W PROBLEMIE NAJKRÓTSZEJ ŚCIEŻKI

Niestety jednak zazwyczaj ani algorytm Dijkstry, ani A* nie mogą w bezpośredni sposób działać na strukturach gry. Do poprawnego działania wymagają specyficznej struktury danych – skierowanego grafu z nieujemnymi wagami.

2.1.1. GRAF

Graf jest pewną matematyczną strukturą składającą się ze zbioru wierzchołków i zbioru połączeń między nimi[5]. Często są one reprezentowane wizualnie przez punkty i połączenia między nimi.

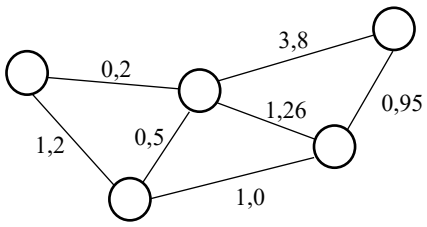


Rysunek 2.1. Przykładowy graf

W pathfindingu wierzchołki grafu zwykle reprezentują pewne obszary albo punkty na mapie czy planszy. Jeśli między takimi dwoma rejonami gry istnieje bezpośrednie przejście, jest ono symbolizowane przez połączenie między odpowiednimi wierzchołkami.

2.1.2. GRAF Z NIEUJEMNYMI WAGAMI

Graf z wagami różni się od zwykłego grafu tym, że do jego połączeń dodaje się pewną numeryczną wartość. W matematyce jest ona nazywana wagą, natomiast w wypadku gier częściej - kosztem. Odpowiada ona kosztowi, jakim obarczone jest przemieszczenie się z jednego wierzchołka na drugi. Im jest on większy, tym trudniej jest przejść pomiędzy danymi wierzchołkami lub zajmuje to więcej czasu.



Rysunek 2.2. Przykładowy graf z wagami

Koszt ten może odpowiadać na przykład czasowi przejścia lub długości ścieżki, ale nic nie stoi na przeszkodzie, żeby był dowolną kombinacją pewnych parametrów.

W takim grafie łatwo też określić całkowity koszt dotarcia z jednego wierzchołka na drugi, z nim nie sąsiadujący. Jest on obliczany jako suma poszczególnych przejść pomiędzy nimi.

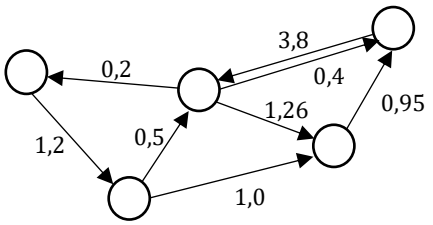
Pomimo że teoria grafów dopuszcza ujemne wagi, to omawiane algorytmy ich nie dopuszczają. Wprawdzie możliwe jest, że w takich przypadkach algorytm zwróci w pewnych wypadkach poprawny wynik, jednak prawdopodobnym jest też, że algorytm Dijkstry czy A* zapętlą się w takim

[4] Ian Millington, *Artificial intelligence for games*, s. 204
[5] Ian Millington, *Artificial intelligence for games*, s. 205

grafie w nieskończoność[6]. Ponadto, jaki sens miałaby ujemna droga czy ujemny czas, które przecież często reprezentuje waga?

2.1.3. GRAF SKIEROWANY

W wielu sytuacjach graf z nieujemnymi wagami jest wystarczający. Jednak łatwo wyobrazić sobie przypadek, w którym przejście pomiędzy pewnymi punktami jest możliwe tylko w jedną stronę. Na przykład z niewielkiego urwiska da się łatwo zeskoczyć, ale wspiąć się na nie, bez dodatkowego sprzętu, jest w zasadzie niemożliwe.



Rysunek 2.3. Przykładowy skierowany graf z wagami

Drugą zaletą użycia grafu skierowanego jest możliwość ustawienia różnych wag w zależności od kierunku przemieszczania się. Kontynuując poprzedni przykład: jeśli podstawimy drabinę pod urwisko, jesteśmy w stanie się po niej wspiąć, ale czas, jaki musimy na to poświęcić, jest znacznie dłuższy niż podróż w drugą stronę.

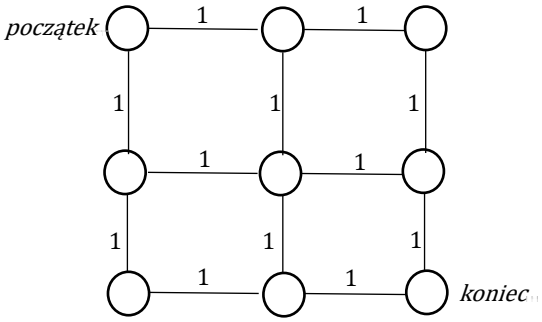
2.2. ALGORYTM DIJKSTRY

Nazwa *algorytmu Dijkstry* pochodzi od nazwiska jego twórcy – Edsgera Wybe'a Dijkstry, holenderskiego matematyka, pioniera informatyki i laureata nagrody Turinga[7] w roku 1972.

Pierwotnie nie miał być to algorytm do wyznaczania najkrótszej ścieżki w dokładnie takim znaczeniu, w jakim rozumiemy to w kontekście gier. Miał on rozwiązywać matematyczny problem z teorii grafów o łudząco podobnej nazwie „najkrótszej ścieżki". Podczas gdy w grach przez ten termin rozumie się znalezienie najkrótszej drogi pomiędzy dwoma punktami, w wypadku teorii grafów jego rozwiązaniem są najkrótsze ścieżki z punktu początkowego do wszystkich innych wierzchołków[8]. Oczywiście chociaż jedną ze składowych jest poszukiwana w wypadku gier ścieżka, to jednak takie podejście byłoby marnotrawstwem zasobów i czasu procesora. Dlatego w tym wypadku zmienia się nieco algorytm Dijkstry tak, żeby szukał i zwracał tylko żądaną ścieżkę.

2.2.1. PROBLEM

Dany jest graf (skierowany z nieujemnymi wagami) oraz dwa z jego wierzchołków zwane początkiem i końcem szukanej ścieżki. Algorytm powinien wygenerować ze wszystkich możliwych ścieżek pomiędzy początkiem a końcem, tę o najniższym koszcie.



Rysunek 2.4. Przykładowy graf, w którym od początku do końca jest kilka ścieżek o równym koszcie

Ścieżek spełniających powyższy warunek może być dowolna liczba. W takim wypadku powinna zostać zwrócona którakolwiek z nich.

[6] Ian Millington, *Artificial intelligence for games*, s. 207-208
[7] Nagroda Turinga przyznawana corocznie od 1966 roku za wybitne osiągnięcia w dziedzinie informatyki przez Association for Computing Machinery. Nazwa została ustanowiona dla uczczenia jednego z twórców współczesnej informatyki, brytyjskiego matematyka Alana Turinga. Nagroda ta bywa czasem określana mianem „informatycznego Nobla".
[8] Ian Millington, *Artificial intelligence for games*, s. 211

2.2.2. ROZWIĄZANIE

W dużym uproszczeniu można powiedzieć, że algorytm Dijkstry działa poprzez rozlewanie się w grafie przez występujące w nim połączenia, od wierzchołków bliżej początkowego do tych położonych dalej. Zapamiętuje on kierunek poruszania się, dzięki czemu po osiągnięciu końcowego wierzchołka jest w stanie odtworzyć drogę, jaką przebył. Sposób, w jaki odbywa się to rozlewanie, gwarantuje, że uzyskana przy odtwarzaniu ścieżka będzie najkrótszą z możliwych[9].

2.2.2.1. PRZETWARZANIE OBECNEGO WĘZŁA

Podczas każdej iteracji przetwarzany jest pewien wierzchołek, zwany dalej obecnym węzłem. Rozważane jest każde wychodzące z niego połączenie wraz z wierzchołkiem, do którego ono prowadzi oraz kosztem całkowitego przejścia z węzła startowego do danego węzła sąsiadującego.

W przypadku, gdy obecnie rozważanym węzłem jest wierzchołek startowy, całkowity koszt sprowadza się do kosztu przejścia przez łączącą oba wierzchołki krawędź. Natomiast w przypadku kolejnych iteracji koszt ten obliczany jest jako suma kosztów przejścia po wszystkich krawędziach prowadzących od wierzchołka początkowego.

2.2.2.2. LISTY WĘZŁÓW

Algorytm Dijkstry przechowuje w pamięci dwie listy węzłów, z którymi do tej pory miał styczność. Pierwsza z nich, zwana listą otwartych, zawiera wszystkie te węzły, które algorytm zobaczył, ale które nie miały jeszcze swojej iteracji. Oznacza to, że w wyniku przetwarzania innego wierzchołka został dla nich obliczony koszt całkowity przejścia i wyznaczony został kierunek, z którego to przejście miało miejsce, ale one same jeszcze nie były przetwarzane. Natomiast na drugiej z nich, zwanej listą zamkniętych, znajdują się wszystkie te węzły, które przeszły już przez swoją iterację.

Każdy wierzchołek grafu może być w takim wypadku w jednym z trzech stanów:

1. Otwartym – wierzchołek jest na liście otwartych i oczekuje na swoją iterację;
2. Zamkniętym – wierzchołek przeszedł już swoją iterację;
3. Nieodwiedzonym – wierzchołek nie został do tej pory osiągnięty.

Algorytm rozpoczyna się od umieszczenia na liście otwartych wierzchołka startowego, natomiast lista zamkniętych jest pusta. Podczas iteracji wierzchołek z najmniejszym do tej pory kosztem całkowitym jest usuwany z listy otwartych i dodawany do listy zamkniętych, a jego każdy sąsiedni wierzchołek jest dodawany do listy otwartych.

W powyższym rozumowaniu może pojawić się jednak pewna komplikacja. Zakłada ono, że za każdym razem, przemieszczając się wzdłuż krawędzi grafu, natrafi się na wierzchołek do tej pory nieodwiedzony. Jednak można równie dobrze trafić na wierzchołek znajdujący się na liście otwartych czy zamkniętych.

2.2.2.3. OBLICZANIE CAŁKOWITEGO KOSZTU DLA LISTY ZAMKNIĘTYCH I OTWARTYCH

Jeśli podczas iteracji dotrzemy do węzła, który został już odwiedzony, to będzie miał on swój całkowity koszt i kierunek, z którego został osiągnięty. Nie można tak po prostu ustawić nowych wartości, ponieważ nadpisałoby to poprzednią pracę wykonaną przez algorytm.

[9] Ian Millington, *Artificial intelligence for games*, s. 211

Zamiast tego należy sprawdzić, czy nowo otrzymana droga jest lepsza od poprzedniej. W tym celu trzeba obliczyć nowy koszt całkowity dla danego węzła i porównać go z kosztem całkowitym zapisanym już dla niego. Jeśli nowa wartość jest większa – a tak będzie w większości wypadków[10] – to można przejść do następnego kroku.

Natomiast jeśli koszt całkowity jest mniejszy, to należy zmienić zarówno koszt całkowity danego węzła, jak i zapisany kierunek. Tak zmieniony węzeł musi znaleźć się na liście otwartych, więc jeśli odwiedzony węzeł o mniejszym koszcie całkowitym jest na liście zamkniętych, trzeba go przenieść z powrotem na listę otwartych.

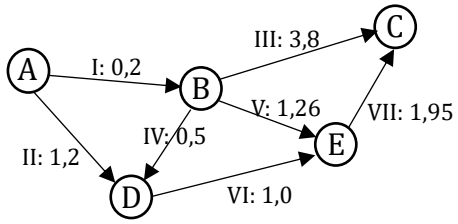
Jednak taka sytuacja w przypadku algorytmu Dijkstry nie ma możliwości zaistnienia z powodu wybierania zawsze węzła o najmniejszym koszcie całkowitym. Jest ona natomiast możliwa w przypadku algorytmu A*, który omawiany będzie w dalszej części pracy.

2.2.2.4. ZAKOŃCZENIE ALGORYTMU

W przypadku pierwotnego wykorzystania algorytmu Dijkstry kończy on swoje działanie, gdy lista otwartych jest pusta, co jest równoznaczne z przeglądnięciem wszystkich osiągalnych wierzchołków.

Jednak w wypadku szukania najkrótszej ścieżki można jego działanie zakończyć wcześniej. Wystarczy, że algorytm zostanie przerwany w momencie, gdy na liście otwartych to właśnie węzeł docelowy jest tym o najmniejszym koszcie całkowitym. Oznacza to także, że odkryto węzeł docelowy w jednej z poprzednich iteracji.

Niestety przerwanie algorytmu w wypadku, gdy pierwszy raz wśród sąsiednich węzłów obecnie iterowanego wierzchołka znajdzie się cel, nie gwarantuje jeszcze tego, że wygenerowana w ten sposób ścieżka będzie najkrótsza.



Rysunek 2.5. Nie zawsze pierwsza ścieżka jest najkrótsza

Rozważmy sytuację z rysunku powyżej, gdzie przyjmiemy, że węzeł A to początek szukanej ścieżki, a węzeł C – jej koniec. Jeżeli przerwalibyśmy algorytm zaraz przy pierwszym napotkaniu węzła C, to otrzymalibyśmy ścieżkę $A \rightarrow B \rightarrow C$ o całkowitym koszcie $0{,}2 + 3{,}8 = 4{,}0$, natomiast najkrótszą ścieżką jest ścieżka $A \rightarrow B \rightarrow E \rightarrow C$, ponieważ jej koszt to $0{,}2 + 1{,}26 + 1{,}95 = 3{,}41$.

Jednak w praktyce programiści często wykorzystują tę drobną metodę przyspieszającą zwrócenie wyniku[11]. Ma to miejsce z kilku powodów: pierwsza droga zazwyczaj jest najkrótsza. A nawet jeśli tak nie jest, to różnica w koszcie całkowitym tych dróg jest zwykle nieznaczna.

2.2.2.5. REKONSTRUKCJA ŚCIEŻKI

Na sam koniec algorytmu otrzymujemy pole końcowe wraz z połączeniem do węzła, z którego do niego doszliśmy. W tak przygotowanej strukturze odtworzenie całej ścieżki jest proste. Wystarczy iterować się w głąb wyznaczonych połączeń z wierzchołka końcowego, aż otrzyma się wierzchołek początkowy. Podczas każdej iteracji należy dodać obecny wierzchołek na początek listy, która reprezentuje szukaną ścieżkę. Jednak mniej kosztowną operacją jest zwykle dodawanie elementu na koniec listy. Wtedy należy tak otrzymaną listę odwrócić.

[10] Ian Millington, *Artificial intelligence for games*, s. 214
[11] Ian Millington, *Artificial intelligence for games*, s. 214-215

2.2.3. PREZENTACJA DZIAŁANIA

Przedstawię działania algorytmu Dijkstry na przykładzie grafu z rysunku 2.5.

W iteracji pierwszej dodane zostaną dwa węzły – B i D – ponieważ sąsiadują one z węzłem A. Z kolei on sam zostanie przeniesiony na listę zamkniętych.

W drugiej iteracji przetwarzane będą węzły sąsiadujące z B – C, D i E – ze względu na to, że węzeł ten ma jak do tej pory najmniejszy koszt całkowity. W tej iteracji zostanie także zaktualizowany koszt całkowity i połączenie dla węzła D, ponieważ droga przez węzeł B jest „tańsza" niż przejście bezpośrednio z A. W tym momencie pierwszy raz dochodzimy do węzła końcowego C, jednak nie można mieć pewności, że otrzymana w ten sposób ścieżka jest „najtańszą" z możliwych.

W kolejnej iteracji jedyne, co się wykona, to przeniesienie badanego węzła D z listy otwartych na listę zamkniętych. Do sąsiadującego z nim węzła E droga $\cdots \rightarrow B \rightarrow D$ byłaby „droższa" niż droga tylko przez węzeł B, więc nie zostanie on uaktualniony.

W tej chwili to właśnie węzeł E ma najmniejszy koszt całkowity i zostanie przebadany jego sąsiad – węzeł C. Na koniec iteracji na liście otwartych najmniejszy koszt całkowity ma końcowy węzeł C, więc można stwierdzić, że otrzymana w ten sposób droga jest najbardziej optymalną.

i	Obecny węzeł	Lista otwartych				Lista zamkniętych			Otrzymana ścieżka
		nr	Węzeł	Koszt	Połączenie	Węzeł	Koszt	Połączenie	
		1.	A	0	–	–	–	–	
1	A	1.	B	0,2	I	A	0	–	
		2.	D	1,2	II				
2	B	1.	D	0,7	IV	A	0	–	$A \rightarrow B \rightarrow C$
		2.	E	1,26	V	B	0,2	I	
		3.	C	3,8	III				
3	D	1.	E	1,26	V	A	0	–	
		2.	C	3,8	III	B	0,2	I	
						D	0,7	IV	
4	E	1.	C	3,41	VII	A	0	–	$A \rightarrow B \rightarrow E \rightarrow C$
						B	0,2	I	
						D	0,7	IV	
						E	1,26	V	

Tabela 2.1 Wyniki poszczególnych iteracji algorytmu Dijkstry i ścieżki, które można by otrzymać w wypadku przerwania algorytmu w danej iteracji dla grafu z rysunku powyżej

2.3. ALGORYTM A*

Szukanie najkrótszej ścieżki to w wypadku gier niemalże synonim *algorytmu A** (wym. *A gwiazdka* lub z ang. *A star*). Dzieje się tak, ponieważ jest on prosty do zaimplementowania, szybki, a dodatkowo daje duże pole do manewru przy optymalizacji.

W przeciwieństwie do algorytmu Dijkstry, A* jest zaprojektowany do nawigacji punkt-punkt, a nie do rozwiązywania problemu najkrótszej ścieżki w rozumieniu teorii grafów.

2.3.1. Problem

Problem i założenia tego algorytmu są identyczne z algorytmem Dijkstry. Potrzebny jest skierowany graf z nieujemnymi wagami. Jego dwa wierzchołki oznaczają początek i koniec szukanej ścieżki. Przy tak zadanych danych algorytm powinien wygenerować najkrótszą możliwą ścieżkę.

2.3.2. Rozwiązanie

Rozwiązanie problemu jest niemalże identyczne jak w przypadku omawianego wcześniej algorytmu Dijkstry. Zmienia się jednak nieco definicja kosztu całkowitego. W tym algorytmie reprezentuje on sumę kosztu dotychczasowej ścieżki i oszacowanego kosztu ścieżki do węzła końcowego.

Wydajność algorytmu w dużej mierze zależy od doboru sposobu owego oszacowania. Jeśli oszacowanie będzie dokładne, to algorytm będzie wydajny, w przeciwnym wypadku algorytm A* może być nawet mniej efektywny od algorytmu Dijkstry[12].

Ze względu na niewielkie różnice w samym algorytmie omówię tylko te jego aspekty, które są nowe w stosunku do algorytmu Dijkstry.

2.3.2.1. Heurystyka

W informatyce *heurystyka* (od starogreckiego *εὑρίσκω* [*heurískō*] – znaleźć, odkryć[13]) jest to technika używana przy znajdowaniu rozwiązań wtedy, gdy zwykłe algorytmy są niewystarczająco wydajne bądź nieznane. Metoda ta nie daje gwarancji znalezienia rozwiązania optymalnego, a często nawet nie daje gwarancji znalezienia rozwiązania w ogóle. Wykorzystywana jest też często do znajdowania bądź oszacowywania pewnych rozwiązań przybliżonych czy cząstkowych, na podstawie których później wyliczane jest rozwiązanie ostateczne[14]. To też ma miejsce w algorytmie A*.

W momencie, gdy w algorytmie Dijkstry podczas iteracji obliczana jest wartość całkowitego kosztu dotarcia do sąsiedniego węzła, w tym algorytmie „wyliczana" jest też pewna nieznana wartość wynikająca z heurystyki (często zwana zmienną heurystyczną), która ma określać pozostały koszt dotarcia z owego sąsiedniego węzła do wierzchołka końcowego[15].

Jako że nie jest możliwe, żeby ten koszt był znany (wtedy przecież znana byłaby również szukana ścieżka), musi być dana jakaś metoda umożliwiająca jego oszacowanie. Oczywiście istnieje ich wiele i są one w dużym stopniu zależne od sposobu prezentacji przestrzeni w grafie.

Jeśli heurystyka zawsze niedoszacowuje pozostałego kosztu, algorytm będzie potrzebował więcej czasu na znalezienie najkrótszej ścieżki. W takim przypadku wyznaczona ścieżka będzie ścieżką o najmniejszym koszcie z możliwych i dokładnie tą samą, która byłaby wyznaczona przez algorytm Dijkstry. Jednakże jeśli choć raz nastąpi przeszacowanie, to nie można już tego zagwarantować[16].

W przeciwnym wypadku, jeśli heurystyka zawsze przeszacowuje wartość pozostałego kosztu, algorytm będzie miał tendencję do generowania krótszych ścieżek, z mniejszą ilością węzłów, nawet jeśli miałyby one większy koszt całkowity. Jednak w ten sposób można wiele zyskać na szybkości algorytmu[12]. Oczywiście jest pewna granica przeszacowania, powyżej której algorytm zaczyna bardzo szybko tracić na wydajności.

---

[12] Ian Millington, *Artificial intelligence for games*, s. 223
[13] Por. *Heuristic* [https://en.wikipedia.org/wiki/Heuristic]; *Εὑρίσκω* [https://pl.wiktionary.org/wiki/εὑρίσκω]
[14] *Heuristic (computer science)* [https://en.wikipedia.org/wiki/Heuristic_(computer_science)]
[15] Por. Mat Buckland, *Programming Game AI by Example* s. 241; Ian Millington, *Artificial intelligence for games*, s. 223-224
[16] Ian Millington, *Artificial intelligence for games*, s. 239-240

2.3.2.1.1. HEURYSTYKA EUKLIDESOWA

Najbardziej naturalnym sposobem oszacowania pozostałej drogi jest obliczenie zwykłej euklidesowej odległości pomiędzy dwoma punktami. Gwarantuje to niemalże zawsze niedoszacowanie zmiennej heurystycznej. W najgorszym przypadku oszacowanie i rzeczywisty koszt będą sobie równe.

2.3.2.1.2. HEURYSTYKA OPARTA NA METRYCE MANHATTAN

Heurystyka ta często wykorzystywana jest w grach, w których przestrzeń opiera się na siatce połączonych ze sobą kwadratów (ang. *grid-like game*)[17]. Odległość w tej metryce określa się wzorem:

$$d_n(A, B) = \sum_{k=1}^{n} |A_k - B_k|,$$

gdzie A i B to punkty w przestrzeni, a n to liczba wymiarów tej przestrzeni. W podanych grach najczęściej przestrzeń jest dwuwymiarowa, w której współrzędne punktów są oznaczane w następujący sposób:

$$P = (P_x, P_y).$$

Sprowadza to powyższy wzór do następującego:

$$d_2(A, B) = |A_x - B_x| + |A_y - B_y|.$$

Głównym powodem używania tej metryki zamiast metryki euklidesowej jest fakt, że nie trzeba obliczać pierwiastka kwadratowego, który jest bardzo kosztowną operacją.

2.3.2.1.3. HEURYSTYKA NULL

Jest to bardzo prosta heurystyka, która w każdym wypadku oszacowuje pozostałą odległość na zero. W takim przypadku koszt całkowity jest z powrotem równy kosztowi ścieżki do badanego węzła, co sprowadza algorytm A* do algorytmu Dijkstry.

2.3.2.2. ZMIANY W ALGORYTMIE WYNIKAJĄCE Z ZASTOSOWANIA HEURYSTYKI

Heurystyka wymusza pewne zmiany w samym algorytmie względem algorytmu Dijkstry.

Po pierwsze wybierany do iteracji jest za każdym razem węzeł o najniższym koszcie całkowitej drogi, a nie jak poprzednio o najmniejszym koszcie drogi umożliwiającej dotarcie do niego. Pozwala to algorytmowi iterować się najpierw przez węzły, które są bardziej obiecujące i wydają się być bliżej szukanego celu. W ten sposób wybierane są takie węzły, które mają relatywnie niski zarówno koszt dotarcia do nich, jak i oszacowany koszt dotarcia od nich do celu.

Po drugie w przypadku natrafienia na węzeł znajdujący się już na liście otwartych bądź zamkniętych możliwe jest, że trzeba będzie poprawić jego wartości. O ile dla algorytmu Dijkstry tylko w przypadku węzłów z listy otwartych należało poprawiać wartości dotychczasowego kosztu ścieżki, o tyle w wypadku algorytmu A* należy to samo zrobić, gdy koszt dotarcia do węzła jest niższy dla węzłów znajdujących się już na liście zamkniętych. Dzieje się tak, ponieważ A* może znaleźć lepszą drogę do węzła, przez który już wcześniej się iterował. W takim przypadku trzeba by tę mniejszą wartość dotychczasowego kosztu rozpropagować przez wszystkie wychodzące połączenia danego węzła, co wymagałoby sporo pracy. Żeby uniknąć tego problemu, wystarczy usunąć dany węzeł z listy zamkniętych i dodać wraz z nowymi wartościami na listę otwartych, żeby mógł być z powrotem przetworzony.

[17] Mat Buckland, *Programming Game AI by Example*, s. 246

Wiele implementacji algorytmu A* kończy jego działanie w momencie, gdy na liście otwartych końcowy węzeł ma najmniejszy całkowity koszt. Jednak, zgodnie z tym, co napisane jest powyżej, może okazać się, że zajdzie potrzeba zmiany wartości w tym węźle z powodu znalezienia krótszej ścieżki. Żeby umożliwić znalezienie tej ścieżki, należy poczekać do momentu, aż węzeł z najmniejszym dotychczasowym kosztem będzie mieć ten koszt co najmniej równy całkowitemu kosztowi węzła końcowego. Jest to w zasadzie taki sam warunek końcowy, jak dla algorytmu Dijkstry, ale przedstawiony z nieco innej perspektywy.

2.3.3. PREZENTACJA DZIAŁANIA

Jako plansza w dalszych rozważaniach przyjęty jest poniższy rysunek. Na planszy koszt przejścia pomiędzy kafelkami liczony jest według wzoru:

$$k = d * f$$

gdzie k to koszt, d to odległość między kafelkami, a f to pewna zmienna zależna od koloru kafelka, która przyjmuje następujące wartości:

- 1 przy przechodzeniu na zielony kafelek;
- 2 przy przechodzeniu na żółty kafelek;
- 0,5 przy przechodzeniu na szary kafelek.

Przejście na niebieski kafelek jest niemożliwe ($f = \infty$). Odległości liczone są zgodnie z metryką euklidesową, na której oparta będzie również heurystyka. Dla ułatwienia obliczeń wartości będą zaokrąglane do dwóch miejsc po przecinku. W nawiasach podano współrzędne poszczególnych pól. Jako startowe przyjęto pole A, natomiast jako końcowe – D.

$\boldsymbol{A(0,0)}$ POCZĄTEK	$B(1,0)$	$C(2,0)$	$\boldsymbol{D(3,0)}$ KONIEC
$E(0,1)$	$F(1,1)$	$G(2,1)$	$H(3,1)$
$I(0,2)$	$J(1,2)$	$K(2,2)$	$L(3,2)$
$M(0,3)$	$N(1,3)$	$O(2,3)$	$P(3,3)$

Rysunek 2.6. Przykładowa plansza gry

Na samym początku nie ma zbyt wielkiego wyboru i z węzła A można przemieścić się tylko na węzeł E. Jest to obarczone kosztem równym $k = 1 * 1 = 1$, funkcja heurystyczna natomiast dla tego węzła zwróci wartość $h = \sqrt{(0-3)^2 + (1-0)^2} = \sqrt{10} \approx 3{,}16$.

W drugiej iteracji badamy węzły I i J. W przypadku węzła I sytuacja jest analogiczna do poprzedniej. Natomiast węzeł J leży po przekątnej węzła E, więc odległość jest równa $d = \sqrt{2} \approx 1{,}41$, a skoro J jest kafelkiem żółtym, to koszt przejścia jest równy $k = 1{,}41 * 2 = 2{,}82$.

W trzeciej iteracji to węzeł I ma najmniejszy koszt całkowity i do listy otwartych dodawani są jego sąsiedzi – M oraz N, natomiast węzeł J znalazł się już wcześniej na liście otwartych. Jednak koszt przejścia $\cdots \rightarrow E \rightarrow I \rightarrow J$ jest równy 4, co daje wyższą wartość niż bezpośrednie przejście z E do J wyliczone w poprzedniej iteracji.

Kolejnym wybranym do iteracji węzłem jest J. W tym kroku na liście otwartych pojawiają się kafelki G, K i O.

Następnie przetwarzany jest węzeł G. W tej iteracji szukany węzeł D pierwszy raz pojawi się na liście otwartych, jednak nie można mieć pewności, że będzie to optymalna trasa do niego, pomimo że jest ona najkrótszą z możliwych.

W kolejnym kroku sprawdzamy sąsiadów węzła H. Okazuje się, że droga $\cdots \rightarrow G \rightarrow H \rightarrow D$ jest mniej kosztowna niż bezpośrednie przejście $\cdots \rightarrow G \rightarrow D$ z poprzedniej iteracji.

Jednak z powodów omówionych wcześniej nie możemy w następnej iteracji przy przetwarzaniu kafelka D przerwać algorytmu. Są na liście otwartych kafelki o mniejszym koszcie przejścia do nich niż on.

W następnych krokach węzły L, K, N oraz M zostaną przeniesione kolejno na listę zamkniętych. Pole C można pominąć, ponieważ wiadomo, że dopóki ma większy koszt dotarcia do niego od startu niż od startu do końcowego kafelka D, to nie da się z niego utworzyć lepszej ścieżki.

W tym momencie wiadomo, że znaleziona wcześniej ścieżka jest „najtańszą” z możliwych.

i	Węzeł	Lista otwartych						Lista zamkniętych					Otrzymana ścieżka
		nr	Węzeł	Koszt	Heu.	Sum.	Poł.	Węzeł	Koszt	Heu.	Sum.	Poł.	
		1.	A	0	-	-	-	-	-	-	-	-	
1	A	1.	E	1	3,16	4,16	A	A	0	-	-	-	
2	E	1.	I	2	3,6	5,6	E	A	0	-	-	-	
		2.	J	3,82	2.83	6,65	E	E	1	3,16	4,16	A	
3	I	1.	J	3,82	2.83	6,65	E	A	0	-	-	-	
		2.	N	4	4.24	8,24	I	E	1	3,16	4,16	A	
		3.	M	4,82	3,61	8,43	I	I	2	3,6	5,6	E	
4	J	1.	G	4,32	1,41	5,73	J	A	0	-	-	-	
		2.	K	5,82	2.24	8,06	J	E	1	3,16	4,16	A	
		3.	N	4	4.24	8,24	I	I	2	3,6	5,6	E	
		4.	M	4,82	3,61	8,43	I	J	3,82	2.83	6,65	E	
5	G	1.	H	4,82	1	5,82	G	A	0	-	-	-	$A \rightarrow E \rightarrow J \rightarrow G \rightarrow D$
		2.	D	7,14	0	7,14	G	E	1	3,16	4,16	A	
		3.	L	5,73	2	7,73	G	I	2	3,6	5,6	E	
		4.	K	5,82	2.24	8,06	J	J	3,82	2.83	6,65	E	
		5.	N	4	4.24	8,24	I	G	4,32	1,41	5,73	J	
		6.	C	7,32	1	8,32	G						
		7.	M	4,82	3,61	8,43	I						
6	H	1.	D	6,48	0	6,48	G	A	0	-	-	-	$A \rightarrow E \rightarrow J \rightarrow G \rightarrow H \rightarrow D$
		2.	L	5,73	2	7,73	G	E	1	3,16	4,16	A	
		3.	K	5,82	2.24	8,06	J	I	2	3,6	5,6	E	
		4.	N	4	4.24	8,24	I	J	3,82	2.83	6,65	E	
		5.	C	7,32	1	8,32	G	G	4,32	1,41	5,73	J	
		6.	M	4,82	3,61	8,43	I	H	4,82	1	5,82	G	
7	D	1.	L	5,73	2	7,73	G	A	0	-	-	-	
		2.	K	5,82	2.24	8,06	J	E	1	3,16	4,16	A	
		3.	N	4	4.24	8,24	I	I	2	3,6	5,6	E	
		4.	C	7,32	1	8,32	G	J	3,82	2.83	6,65	E	
		5.	M	4,82	3,61	8,43	I	G	4,32	1,41	5,73	J	
								H	4,82	1	5,82	G	
								D	6,48	0	6,48	G	
8	L	1.	C	7,32	1	8,32	G	A	0	-	-	-	
9	K							E	1	3,16	4,16	A	
10	N							I	2	3,6	5,6	E	
11	M							J	3,82	2.83	6,65	E	
								G	4,32	1,41	5,73	J	
								H	4,82	1	5,82	G	
								D	6,48	0	6,48	G	
								L	5,73	2	7,73	G	
								K	5,82	2.24	8,06	J	
								N	4	4.24	8,24	I	
								M	4,82	3,61	8,43	I	

Tabela 2.2. Wyniki poszczególnych iteracji algorytmu A i ścieżki, które można by otrzymać w wypadku przerwania algorytmu w danej iteracji dla planszy z rysunku Rysunek 2.6. Przykładowa plansza gry*

2.4. Implementacja

W mojej implementacji posłużyłem się omówionym powyżej algorytmem A*, a niewielka zmiana, jaką zastosowałem, wynikała z chęci przyspieszenia działania algorytmu w momencie, gdy pole początkowe i końcowe leżą koło siebie.

2.4.1. Funkcje heurystyczne

Omawianie zacznę od najprostszej części, czyli od funkcji heurystycznych.

2.4.1.1. Heurystyka euklidesowa

```
1.  public static float EuclideanDistance(MapField from, MapField to) {
2.     return (float)Sqrt((from.MapPosition.X - to.MapPosition.X) *
3.                        (from.MapPosition.X - to.MapPosition.X) +
4.                        (from.MapPosition.Y - to.MapPosition.Y) *
5.                        (from.MapPosition.Y - to.MapPosition.Y));
6.  }
```

Funkcja `EuclideanDistance` oblicza odległość zgodnie z metryką euklidesową.

W tym przypadku nie zastosowałem wbudowanej funkcji do obliczania potęgi ze względu na jej niższą wydajność w porównaniu do zastosowanego wpisania wprost w kod obliczeń.

2.4.1.2. Heurystyka Manhattan

```
1.  public static float ManhattanDistance(MapField from, MapField to) {
2.     return Abs(from.MapPosition.X - to.MapPosition.X) +
3.            Abs(from.MapPosition.Y - to.MapPosition.Y);
4.  }
```

Funkcja `ManhattanDistance` wylicza odległość zgodnie z metryką Manhattan.

2.4.1.3. Heurystyka null

```
1.  public static float NullDistance(MapField from, MapField to) {
2.     return 0f;
3.  }
```

Funkcja `NullDistance` w każdej sytuacji zwraca wartość 0. Zastosowanie jej sprowadza zastosowany algorytm do Dijkstry.

2.4.2. Klasa reprezentująca węzeł grafu wraz z połączeniami

```
1.  private class PathFindingNode : IComparable<PathFindingNode> {
2.     internal PathFindingNode(MapField This) { this.This = This; }
3.
4.     public MapField This { get; }
5.     public PathFindingNode Parent { get; set; } = null;
6.
7.     public float CostFromStart { get; set; } = 0f;
8.     public float CostToEnd { get; set; } = float.MinValue;
9.
10.    public float AllCost => CostFromStart + CostToEnd;
11.
12.    public int CompareTo(PathFindingNode two) {
13.       if ( AllCost < two.AllCost ) return -1;
14.       if ( AllCost == two.AllCost ) return 0;
15.       return 1;
16.    }
17. }
```

Powyższa klasa jest klasą prywatną, ponieważ jej zastosowanie w innym kontekście niż szukanie ścieżki nie ma sensu. Konstruktor przyjmuje jako argument pole, które ma być reprezentowane przez dany węzeł. Przypisywane jest ono do własności `This`. Właściwość `Parent` może być zmienna

w czasie i reprezentuje ścieżkę do węzła, z którego dokonano przejścia do obecnego węzła. Wartością domyślną jest `null`, ponieważ nie jest to obowiązkowe pole, na przykład węzeł początkowy nigdy nie będzie mieć węzła-rodzica. Klasa ma też dwie właściwości odpowiadające kosztowi dotarcia od początku do obecnego węzła – `CostFromStart`, wartość domyślna to: 0 – oraz przewidywanemu kosztowi dotarcia do węzła końcowego – `CostToEnd`, wartość domyślna to: `float.MinValue`. W przypadku drugim jest potrzebna taka wartość, żeby nigdy nie zaszła sytuacja, w której droga od węzła początkowego czy węzłów, które mają zostać pominięte z innych przyczyn, osiągnięta w danej iteracji, była krótsza. Klasa zawiera też własność `AllCost`, która jest tylko do odczytu i jest obliczana jako suma kosztów przechowywanych w węźle. Dodatkowo klasa ta zawiera przeciążone operatory równości == i !=, które sprawdzają, czy własność `This` obu porównywanych węzłów jest tym samym polem na mapie. Przeładowana jest także metoda `CompareTo` wykorzystywana przy porównywaniu podczas algorytmu sortowania. Połączenia między sąsiednimi węzłami są osiągalne poprzez właściwość `This.Neighbour` umożliwiającą odwołanie się do dowolnego pola mapy poprzez podanie jego położenia względem obecnego pola. W tym kontekście jednak wykorzystywany jest udostępniany przez tę właściwość iterator, który iteruje się tylko po najbliższych ośmiu sąsiadach danego węzła. Istnienie połączenia jest sprawdzane przez własność `IsAvaliable` z klasy `MapField`, która na podstawie wewnętrznego stanu klasy zwraca, czy na danym polu można w tym momencie stanąć albo przez nie przejść, czy nie.

2.4.3. Funkcja rekonstruująca ścieżkę

```
1.  private static IList<MapField> ReconstructPath(PathFindingNode node) {
2.     List<MapField> path = new List<MapField>();
3.
4.     while ( true ) {
5.        path.Add(node.This);
6.        if ( node.Parent == null ) break;
7.        node = node.Parent;
8.     }
9.
10.    path.Reverse();
11.    return path;
12. }
```

Funkcja `ReconstructPath` rekonstruuje ścieżkę, w sposób opisany w rozdziale 2.2.2.5.

2.4.4. Funkcja algorytmu A*

```
1.  public static IList<MapField> AStar(MapField from, MapField to, heuristicFunc heuristic,
2.                                      params MapField[] forbidenFields) {
3.
4.     // ktores z pol nie istnieje
5.     if ( (from == null) || (to == null) ) throw new NullReferenceException();
6.
7.     // pole jest niedostepne
8.     if ( !to.IsAvaliable ) throw new FieldNotAvaliableException();
9.
10.    // pola from i to leza kolo siebie
11.    if ( from.Neighbour.Contains(to) ) return new List<MapField>(2) { from, to, };
12.
13.    //Lita pol do przeszukania
14.    List<PathFindingNode> openList = new List<PathFindingNode> { new PathFindingNode(from) };
15.
16.    //Lista przeszukanych pol
17.    List<PathFindingNode> closeList = new List<PathFindingNode>(
18.                  forbidenFields.Select(forbidenField => new PathFindingNode(forbidenField)));
19.
20.    bool reachedGoal = false;
21.
```

```
22.    //dopoki jakiekolwiek pole moze zostac jeszcze przebadane
23.    while ( openList.Count != 0 ) {
24.       PathFindingNode current = openList.RemoveAtAndGet(0);
25.       closeList.Add(current);
26.
27.       if ( heuristic == NullDistance && current.This == to ) {
28.          return ReconstructPath(current);
29.       } else if ( current.This == to ) { // znaleziono pole do ktorego dazylismy
30.          reachedGoal = true;
31.       } else if ( reachedGoal ) {
32.          PathFindingNode endNode = openList.Concat(closeList)
33.                                            .First(node => node.This == to);
34.
35.          float minValInOpenList = openList.Select(node => node.CostFromStart)
36.                                           .Min();
37.          if ( minValInOpenList < endNode.CostFromStart ) return ReconstructPath(endNode);
38.       }
39.
40.       foreach ( PathFindingNode neighbourNode in current.This.Neighbour
41.                                             .Where(neighbour => neighbour.IsAvaliable)
42.                                             .Select(neighbour => new PathFindingNode(neighbour) {
43.                                                Parent = current,
44.                                                CostFromStart = current.CostFromStart +
45.                                                   EuclideanDistance(current.This, neighbour) *
46.                                                   current.This.Cost,
47.                                                CostToEnd = heuristic(neighbour, to)
48.                                             }) ) {
49.          // Zmiana parametrow sciezki jesli droga do sasiada
50.          // jest krotsza z obecnego pola niz ustalona wczesniej
51.          if ( heuristic != NullDistance && closeList.Contains(neighbourNode) ) {
52.             PathFindingNode oldNode = closeList[closeList.IndexOf(neighbourNode)];
53.             if ( neighbourNode.CostFromStart < oldNode.CostFromStart ) {
54.                closeList.Remove(oldNode);
55.                openList.Add(neighbourNode);
56.             }
57.          } else if ( openList.Contains(neighbourNode) ) {
58.             PathFindingNode oldNode = openList[openList.IndexOf(neighbourNode)];
59.             if ( neighbourNode.CostFromStart < oldNode.CostFromStart ) {
60.                oldNode.CostFromStart = neighbourNode.CostFromStart;
61.                oldNode.Parent = neighbourNode.Parent;
62.             }
63.          } else {
64.             openList.Add(neighbourNode);
65.          }
66.       }
67.
68.       openList.Sort();
69.    }
70.
71.    //jesli nie dotarto do zadanego celu
72.    throw new FieldNotAvaliableException();
73. }
```

Na koniec omówię funkcję `AStar`, która stricte wyznacza najkrótszą ścieżkę. Przyjmuje ona cztery argumenty:

- `MapField from`, `MapField to` – są to odpowiednio: pole początkowe i pole docelowe, między którymi szukamy ścieżki;
- `heuristicFunc heuristic` – jest to funkcja heurystyczna, która ma być używana w algorytmie;
- `params MapField[] forbidenFields` – jest to lista pól, które mają być pominięte przez algorytm z jakichś zewnętrznych, nieznanych powodów. Domyślnie jest ona pusta. Jej elementy mogą w wywołaniu być podawane jako kolejne argumenty po prostu po przecinku

albo może zostać przekazana tablica zawierająca te pola. Może również nie zostać podane żadne pole. Dzięki temu funkcja `AStar` jest funkcją o zmiennej liczbie parametrów.

Na pewne wyjaśnienie zasługuje tutaj typ `heuristicFunc`, którego pełna deklaracja jest następująca: `public delegate float heuristicFunc(MapField from, MapField to)`. Jest to tak zwany delegat[18]. W języku C# delegat odpowiada w pewnym sensie wskaźnikowi na funkcję znanemu z języka C++. Pozwala metodom przyjmować inne funkcje jako parametry, tak jak w mojej implementacji algorytmu A*. Jednak w przeciwieństwie do wskaźnika na funkcję jest on po pierwsze bezpieczny typowo i zarządzany przez odśmiecacz, a po drugie umożliwia podpięcie kilku metod pod jeden delegat (przy pomocy przeładowanych operatorów + i +=), co jest wykorzystywane w wypadku wbudowanej w język C# obsługi zdarzeń. W tym przypadku funkcje są wykonywane w kolejności, w jakiej zostały dołączone do delegatu, a jeśli delegat wymaga zwracanej wartości, zwracana jest ta z ostatniego wywołania. Ważnym do odnotowania jest też fakt, że o ile przy normalnym przeciążaniu funkcji zwracany typ nie ma znaczenia, o tyle w wypadku delegatu jest on jego integralną częścią. Do delegatu mogą być przypisywane dowolne dostępne funkcje, zarówno statyczne, jak i należące do konkretnej instancji klasy.

Na początku algorytm sprawdza poprawność przekazanych danych. Jeśli napotka jakąś niezgodność z oczekiwaniami, rzuci odpowiedni wyjątek. Następnie sprawdzane jest, czy przypadkiem pola początkowe i końcowe nie leżą koło siebie. Jeśli tak jest, zwracana jest lista zawierająca te dwa pola. Zaimplementowałem ten warunek w celu uproszczenia pewnych sytuacji. Możliwym jest bowiem, że zajdzie przypadek, w którym istnieć będzie szybsze połączenie niż przejście bezpośrednio między polami, ale uznałem, że jest on wystarczająco rzadki, aby móc go pominąć. Następnie tworzona jest lista otwartych, na której początkowo znajduje się tylko węzeł startowy oraz lista zamkniętych, na której znajdują się wszystkie pola z tablicy `forbidenFields`. W kolejnym etapie tworzona jest zmienna `reachedGoal`, która odpowiada, czy w trakcie działania algorytm dotarł już do pola końcowego. Jest to potrzebne, aby zaimplementować odpowiednie kończenie algorytmu. Następnie algorytm wchodzi do pętli, na początku której pobiera pierwszy element z listy otwartych do przeprocesowania. Jeśli jest to węzeł końcowy oraz heurystyka sprowadza algorytm do algorytmu Dijkstry, to rekonstruuje on ścieżkę z osiągniętego węzła końcowego i zwraca ją, natomiast jeśli heurystyka jest inna, zmienia się wartość zmiennej `reachedGoal` na `true`. Natomiast jeśli w którejś z poprzednich iteracji dotarto do węzła końcowego, to sprawdza, czy wartość kosztu dotarcia do obecnego węzła jest mniejsza niż koszt dotarcia do węzła końcowego. Jeśli tak jest, algorytm kończy swoje działanie, rekonstruuje ścieżkę z węzła końcowego i zwraca ją. W przeciwnym wypadku algorytm procesuje wszystkich sąsiadów obecnego węzła. W wewnętrznej pętli `foreach` najpierw tworzony jest nowy węzeł na podstawie informacji z węzła-rodzica. Odległość od początku ścieżki jest obliczana jako suma odległości od początku ścieżki węzła-rodzica i odległości euklidesowej miedzy nimi dwoma. Wartość heurystyczna szacowana jest na podstawie funkcji heurystycznej przekazanej jako argument. Następnie sprawdzane jest, czy taki węzeł nie znajduje się na którejś z list. Jeśli zostanie znaleziony, to jeśli koszt dotarcia do niego z węzła startowego jest wyższy niż koszt dotarcia z przejściem przez węzeł, z którego właśnie został osiągnięty, to węzeł znajdujący się na liście zostanie usunięty i zamieniony na właśnie stworzony. Jeśli nie zostanie znaleziony, będzie po prostu dodany do listy otwartych. W tym momencie kończy się wewnętrzna pętla. Na koniec lista otwartych

[18] *Delegates* [https://docs.microsoft.com/pl-pl/dotnet/articles/csharp/programming-guide/delegates/]

zostaje posortowana, żeby móc wyjąć z niej odpowiedni element. Pętla powtarza się tak długo, dopóki lista otwartych zawiera jakieś elementy. Jeśli pętla skończy się w „naturalny” sposób, oznacza to, że nie znaleziono ścieżki i algorytm w tym momencie rzuca odpowiedni wyjątek.

2.5. TESTOWANIE

Postanowiłem wykonać kilka testów wydajnościowych i jakościowych, żeby porównać, jak zachowuje się algorytm w zależności od metryki, na jakiej operuje.

W algorytmie nie było dokonanych żadnych zmian w jego logice. Dodałem zaledwie jeden argument, którym był obiekt struktury `AStarDiagnostic`. Zbierał on pewne dane diagnostyczne w trakcie działania algorytmu po to, żeby później móc je zapisać w odpowiednim formacie do pliku. Sam zapis realizowałem oczywiście po zakończeniu odmierzania czasu, już poza funkcją `AStar`, żeby jak najmniej zaburzać wynik pomiaru czasu.

Wszystkie testy były uruchomione po skompilowaniu przez *Visual Studio 2017 RC* w konfiguracji *Release* przy włączonej optymalizacji kodu, na moim komputerze o następującej specyfikacji:

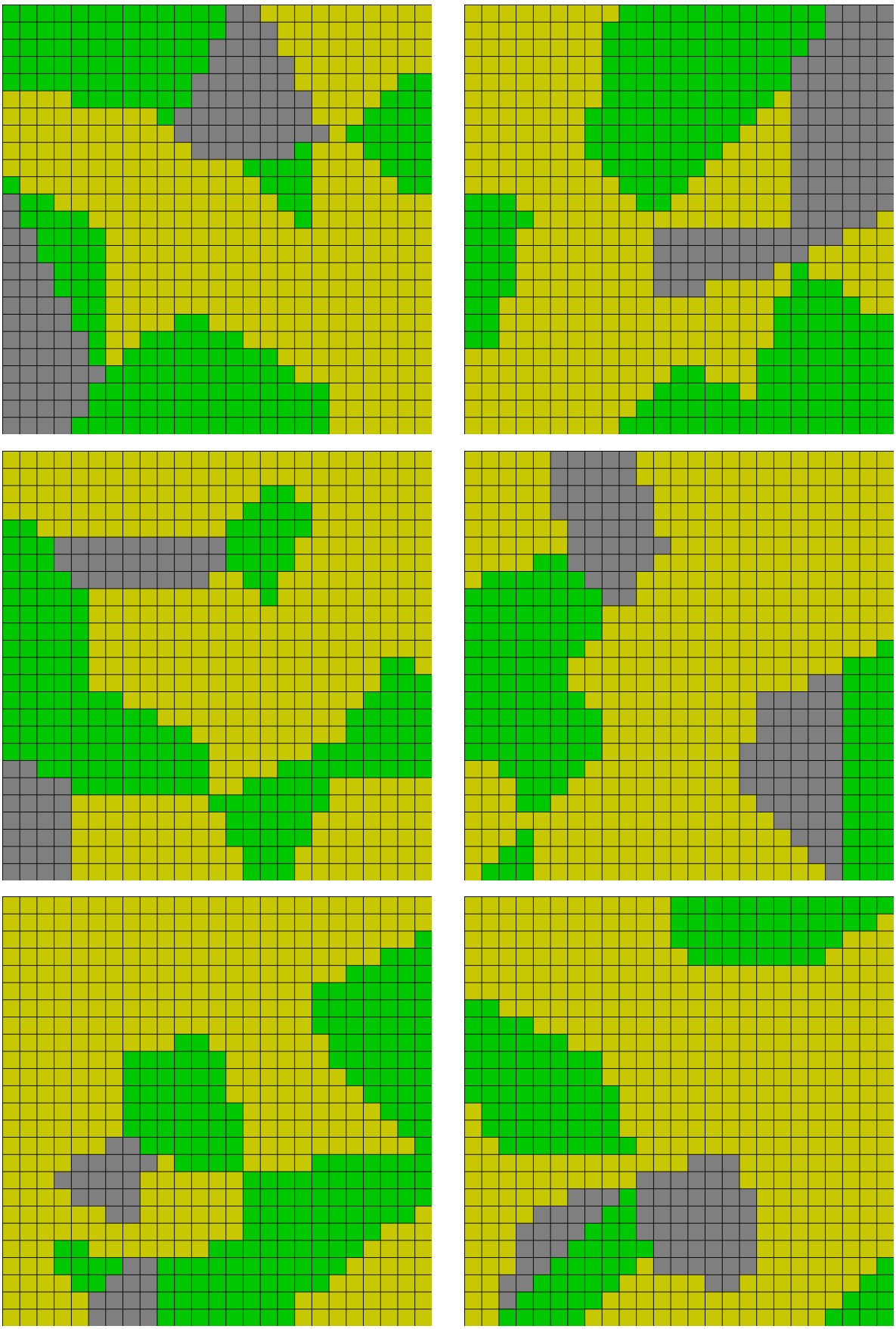
Procesor:	Intel® Core™ $i7-3630QM$
Taktowanie procesora:	$2{,}4\ GHz$
Pamięć RAM:	$8\ GB$
System operacyjny:	Microsoft Windows 10

Uruchomiłem testy na dwa sposoby. Za pierwszym razem były one uruchomione na 200 planszach o wymiarach 25×25. Mapy były inicjalizowane raz na test, po czym uruchamiany był trzykrotnie algorytm A*, po razie dla każdej zaimplementowanej metryki. Na każdej mapie wyszukiwał on drogę między dolnym lewym a górnym prawym rogiem planszy, co dalej będę traktował jako długą ścieżkę. Natomiast za drugim razem testy były uruchomione na 20 planszach o wymiarach 25×25. Mapy były inicjalizowane raz na test, po czym algorytm A* uruchamiany był trzydziestokrotnie, po dziesięć razy dla każdej zaimplementowanej metryki. W każdej iteracji na danej planszy, która obejmowała wyznaczenie ścieżki dla wszystkich trzech metryk, wyznaczane były dwa pseudolosowe, dostępne pola, między którymi algorytmy wyznaczały ścieżki, które dalej będę traktował jako krótkie. Daje to w sumie 1200 pojedynczych testów, po 400 na każdą metrykę.

Odchylenie standardowe σ, o jakim będę mówił w tym rozdziale, jest wyliczane ze wzoru:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N}(x_i - \bar{x})}{N}},$$

gdzie x_i to i-ta próbka, $\bar{x}$ to średnia arytmetyczna ze wszystkich próbek, a N to liczba wszystkich próbek. Jest to tak zwane odchylenie standardowe populacji. Wszystkie dane będą zaokrąglone do dwóch miejsc po przecinku.



Rysunek 2.7. Przykładowe sześć plansz, na których były uruchomione testy

2.5.1. TESTY WYDAJNOŚCIOWE

W testach wydajnościowych mierzyłem takie wartości jak:

- czas wyznaczenia ścieżki – użyłem w tym celu klasy `Stopwatch` z przestrzeni nazw `System.Diagnostics` oferującą bardzo prosty interfejs do mierzenia czasu, który zapisywałem później do pliku w ms;
- liczbę iteracji – użyłem w tym celu zmiennej typu `uint`, która była inkrementowana w każdym obiegu pętli.

Algorytm	Metryka	Średni czas wykonania $[ms]$	Odchylenie standardowe $[ms]$	Średnia liczba iteracji	Odchylenie standardowe
Algorytm A*	Metryka euklidesowa	125,24	48,96	386,74	89,23
	Metryka Manhattan	62,49	56,05	272,2	169,11
Algorytm Dijkstry		2335,51	585,53	6442,04	1212,63

Tabela 2.3. Średnie czasy wykonania i liczba iteracji oraz odchylenia standardowe tych wielkości dla długich ścieżek

Po wygenerowaniu danych w celach porównawczych obliczyłem średnie stosunki pomiędzy różnymi metrykami dla czasów wykonania i liczby iteracji. Warto odnotować tutaj fakt, że mimo szukania najkrótszej ścieżki między dwoma tymi samymi punktami na różnych planszach wyznaczone ścieżki miały oczywiście różną długość i stąd mogą brać się pewne odchylenia w wartościach.

W tabelach 2.4.-2.7. zastosowano następujące oznaczenia:

- A* E – algorytm A* z wykorzystaniem metryki euklidesowej;
- A* M – algorytm A* z wykorzystaniem metryki Manhattan;
- AD – algorytm Dijkstry.

	A* E	A* M	AD
A* E	X	0,46 [0,28]	20,35 [5,9]
A* M	3,13 [2,15]	X	66,46 [54,09]
AD	0,05 [0,01]	0,02 [0,02]	X

	A* E	A* M	AD
A* E	X	0,67 [0,32]	16,97 [2,23]
A* M	1,92 [1,11]	X	33,68 [23,12]
AD	0,06 [0,01]	0,04 [0,02]	X

Tabela 2.4. Tabele średnich stosunków pomiędzy różnymi metrykami dla czasów wykonania (po prawej) i liczby iteracji (po lewej) dla długich ścieżek. Wyliczane są jako stosunki wartości pionowej do poziomej. W nawiasie kwadratowym podano odchylenie standardowe.

Dla pełności obrazu warto dodać, że algorytm A* przy metryce euklidesowej okazał się tylko w 7,5% przypadków szybszy niż przy użyciu metryki Manhattan. Natomiast w przeciwną stronę, przy wykorzystaniu metryki Manhattan, czas wykonania był co najmniej dwa razy szybszy w aż 63,5% przypadków. Ponadto tylko w 17% przypadków wykorzystanie metryki euklidesowej sprowadziło się do przeprowadzenia mniejszej liczby iteracji. W żadnym przypadku lepszy nie okazał się algorytm Dijkstry.

Przedstawienie tabeli ze średnimi czasami wykonania i liczbą iteracji dla krótkich ścieżek nie miałoby sensu, ponieważ są one generowane pseudolosowo i za każdym razem odległość między polem początkowym i końcowym może być inna. Dlatego ograniczę się jedynie do tabel ze średnimi stosunkami pomiędzy różnymi metrykami dla czasów wykonania i liczby iteracji. Warto zaznaczyć

tutaj, że kilka razy zostały wylosowane pola ze sobą sąsiadujące, co można było poznać po zerowej liczbie iteracji. Ze względu na fakt, że algorytm nie jest wtedy uruchamiany, dane te odrzuciłem z obliczeń.

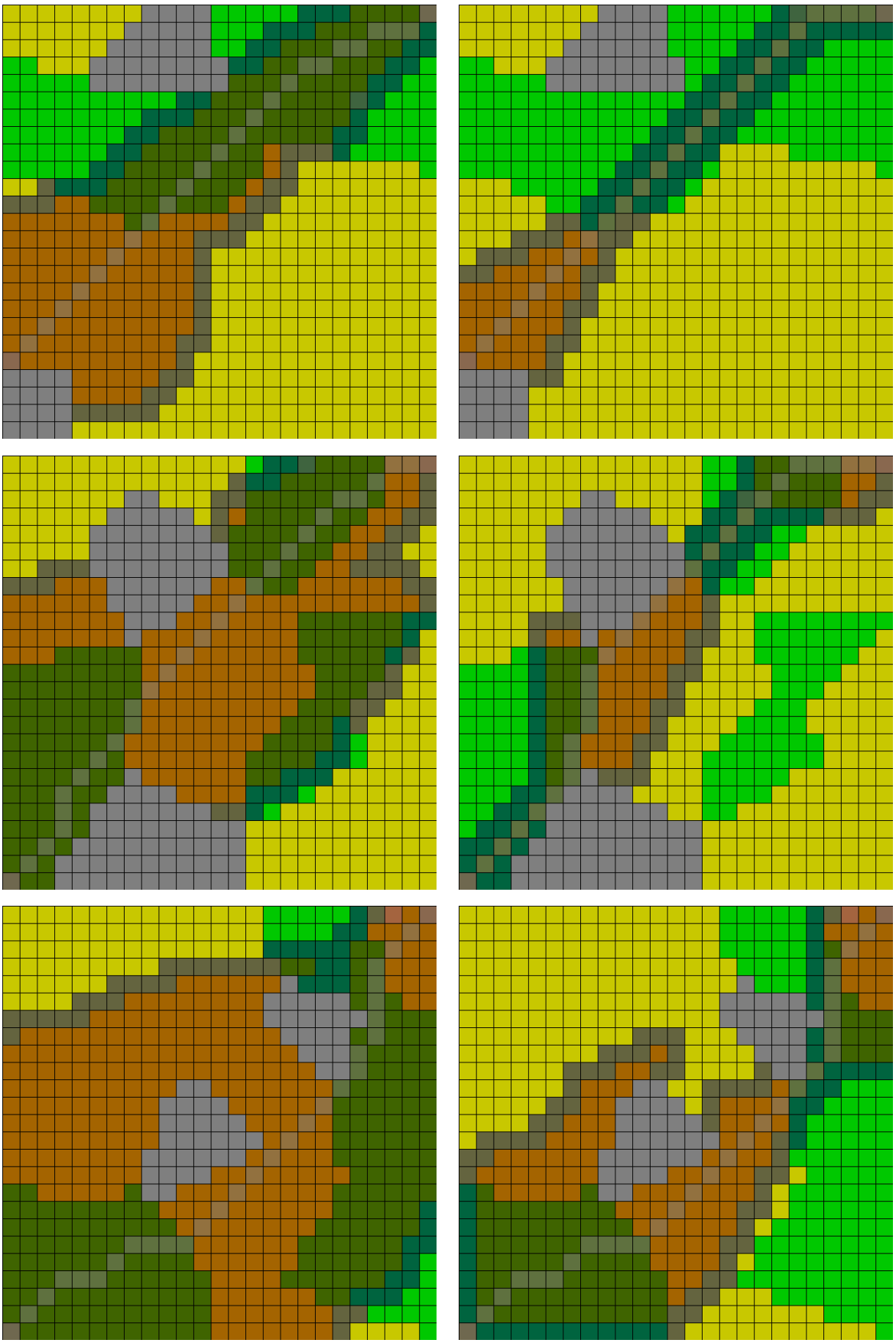
	A* E	A* M	AD
A* E	X	0,51 [0,22]	39,95 [38,85]
A* M	2,45 [1,48]	X	103,4 [118,39]
AD	0,05 [0,07]	0,03 [0,05]	X

	A* E	A* M	AD
A* E	X	0,56 [0,69]	13,2 [5,37]
A* M	1,57 [0,52]	X	21,87 [14,63]
AD	0,09 [0,06]	0,03 [0,07]	X

Tabela 2.5. Tabele średnich stosunków pomiędzy różnymi metrykami dla czasów wykonania (po prawej) i liczby iteracji (po lewej) dla krótkich ścieżek. Wyliczane są jako stosunki wartości pionowej do poziomej. W nawiasie kwadratowym podano odchylenie standardowe.

Dla pełności obrazu warto dodać, że algorytm A* przy metryce euklidesowej okazał się tylko w 3,06% przypadków szybszy niż przy użyciu metryki Manhattan. Natomiast w drugą stronę, przy wykorzystaniu metryki Manhattan, w aż 51,53% przypadków czas wykonania był szybszy co najmniej dwa razy. Ponadto tylko w 2,04% przypadków wykorzystanie metryki euklidesowej sprowadziło się do przeprowadzenia mniejszej liczby iteracji. W żadnym przypadku lepszy nie okazał się algorytm Dijkstry.

Z powyższych danych jasno wynika, że pewne ukierunkowanie algorytmu na cel, jak to ma miejsce w wypadku algorytmu A*, daje bardzo duże benefity, zarówno czasowe, jak i ilościowe. Dodatkowo można zauważyć, że zazwyczaj szybsze będzie wykorzystanie metryki Manhattan, która nie musi w każdej iteracji wyliczać kilkukrotnie kosztownego obliczeniowo pierwiastka kwadratowego. Można zaobserwować także, że dla dłuższych ścieżek zysk ten jest większy. Widać tutaj także wyraźnie różnice pomiędzy metryką przeszacowującą (Manhattan) a niedoszacowującą (euklidesowa), o czym pisałem w rozdziale 2.3.2.1. Nie ma również żadnego zaskoczenia w wynikach algorytmu Dijkstry, który okazał się najbardziej kosztowny czasowo i badający największą liczbę węzłów.



*Rysunek 2.8. Reprezentacja listy otwartych i zamkniętych dla algorytmu A**
przy zastosowaniu metryki euklidesowej (po lewej) i metryki Manhattan (po prawej).
Półprzezroczystym niebieskim zaznaczone są pola do przebadania znajdujące się na liście otwartych.
Półprzezroczystym czerwonym zaznaczone są pola przebadane znajdujące się na liście zamkniętych.
Półprzezroczystym białym zaznaczono wyznaczoną przez algorytm ścieżkę.

2.5.2. TESTY JAKOŚCIOWE

W testach jakościowych mierzyłem takie wartości jak:

- koszt wyznaczonej ścieżki – użyłem w tym celu zmiennej typu `float`, która była równa kosztowi dotarcia do pola końcowego;
- długości wyznaczonej ścieżki – użyłem w tym celu zmiennej typu `uint`, która była inkrementowana podczas rekonstrukcji ścieżki.

W tym przypadku nie ma zasadności uśrednianie kosztu wyznaczonej ścieżki czy jej długości, ponieważ wartości te są ściśle związane z tym, jakie pola zostały wylosowane i jak daleko one od siebie leżą. Jednak nic nie stoi na przeszkodzie, żeby policzyć średnie stosunki pomiędzy różnymi metrykami dla długości i kosztu ścieżki przy zastosowaniu różnych metryk.

	A* E	A* M	AD
A* E	X	1,004 [0,01]	1 [0]
A* M	0,996 [0,01]	X	0,996 [0,01]
AD	1 [0]	1,004 [0,01]	X

	A* E	A* M	AD
A* E	X	1,003 [0,06]	1 [0]
A* M	0,999 [0,06]	X	0,999 [0,06]
AD	1 [0]	1,003 [0,06]	X

Tabela 2.6. Tabele średnich stosunków pomiędzy różnymi metrykami dla kosztu (po prawej) i długości (po lewej) wyznaczonej ścieżki dla długich ścieżek. Wyliczane są jako stosunki wartości pionowej do poziomej. W nawiasie kwadratowym podano odchylenie standardowe.

Dla pełności obrazu warto dodać, że algorytm A* przy metryce Manhattan wyznaczył tylko w 7% przypadków dłuższą, a w 4% przypadków krótszą ścieżkę niż przy użyciu metryki euklidesowej. Natomiast w 33% przypadków koszt wyznaczonej ścieżki okazał się wyższy niż dla algorytmu Dijkstry. Algorytmy Dijkstry i A* przy zastosowaniu metryki euklidesowej w obu wypadkach okazały się równoważne.

	A* E	A* M	AD
A* E	X	1,003 [0,01]	1 [0]
A* M	0,997 [0,01]	X	0,997 [0,01]
AD	1 [0]	1,003 [0,01]	X

	A* E	A* M	AD
A* E	X	1 [0,01]	1 [0]
A* M	1 [0,01]	X	1 [0,01]
AD	1 [0]	1 [0,01]	X

Tabela 2.7. Tabele średnich stosunków pomiędzy różnymi metrykami dla kosztu (po prawej) i długości (po lewej) wyznaczonej ścieżki dla krótkich ścieżek. Wyliczane są jako stosunki wartości pionowej do poziomej. W nawiasie kwadratowym podano odchylenie standardowe.

Dla pełności obrazu warto dodać, że algorytm A* przy metryce Manhattan wyznaczył tylko w 0,51% przypadków dłuższą, a także w 0,51% przypadków krótszą ścieżkę niż przy użyciu metryki euklidesowej. Natomiast w 15,81% przypadków koszt wyznaczonej ścieżki okazał się wyższy niż dla algorytmu Dijkstry. Algorytmy Dijkstry i A* przy zastosowaniu metryki euklidesowej w obu wypadkach okazały się równoważne.

Na podstawie powyższych danych łatwo wywnioskować, że im krótsze ścieżki, tym częściej wykorzystanie metryki Manhattan da ten sam wynik co zastosowanie algorytmu Dijkstry. Jednak w wypadku testów jakościowych można też zaobserwować, że wszystkie różnice są bardzo niewielkie i pomiędzy tymi wersjami algorytmu sprowadzają się głównie do różnic wydajnościowych.

3. ALGORYTM STADA

Sporo gier wykorzystuje pewne zgromadzenia obiektów, które mają wykazywać zachowania stadne. Mogą to być na przykład stada ptaków, ławice ryb, roje pszczół, watahy wilków czy spore grupy przeciwników. Aby nadać tym zachowaniom nieco realizmu, wykorzystuje się algorytm stada, z angielskiego zwany *boids*. Został on stworzony w roku 1986 przez Craiga Reynoldsa i zaprezentowany rok później[19].

3.1. PROBLEM

Dana jest pewna grupa obiektów, zwanych dalej agentami, której trzeba nadać pewne realistyczne zbiorowe zachowania.

3.2. ROZWIĄZANIE

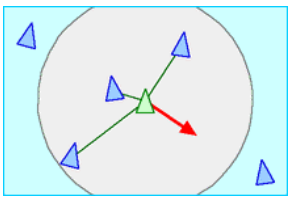
Zaproponowane przez Reynoldsa rozwiązanie jest bardzo proste i pierwotnie opierało się na trzech regułach. Dopiero po pewnym czasie od opublikowania implementacji została dodana czwarta[20].

Rozwiązanie to polega na stosowaniu poniższych reguł przy każdym cyklu (uaktualnieniu), w którym agenci za każdym razem sprawdzają środowisko, w jakim w danej chwili przebywają, więc takie stado może bardzo szybko reagować na zachodzące w otoczeniu zmiany.

Ponadto w podstawowej wersji algorytm stada jest algorytmem bezstanowym[21]. Do podjęcia działania agentowi wystarczy jedynie wiedza o jego najbliższym otoczeniu. Dzięki temu zminimalizowana jest ilość pamięci wymaganej do działania algorytmu.

3.2.1. REGUŁA ROZDZIELNOŚCI

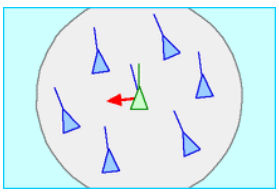
Reguła rozdzielności (ang. *separation*) ma zapobiegać tworzeniu się tłumu w jednym miejscu. Nakazuje ona agentom, aby zachowywali oni od siebie pewną określoną odległość.



Rysunek 3.1.[17] Przedstawienie reguły rozdzielności

3.2.2. REGUŁA WYRÓWNANIA

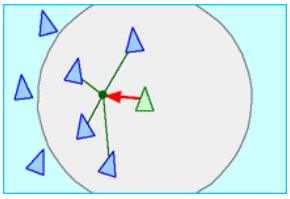
Reguła wyrównania (ang. *alignment*) daje agentowi możliwość zmiany kierunku i prędkości swego przemieszczania tak, że może on dostosowywać te parametry do innych agentów przebywających w jego pobliżu.



Rysunek 3.2.[17] Przedstawienie reguły wyrównania

3.2.3. REGUŁA SPÓJNOŚCI

Reguła spójności (ang. *cohesion*) nakazuje agentom poruszanie się w kierunku średniego położenia agentów w ich najbliższym otoczeniu.



Rysunek 3.3.[17] Przedstawienie reguły spójności

[19] Craig Reynolds, *Boids* [http://www.red3d.com/cwr/boids/].
[20] Por. Mark DeLoura, *Game Programming Gems*, s. 306; Krzysztof Wardziński, *Przegląd algorytmów sztucznej inteligencji stosowanych w grach komputerowych*, s 251.
[21] Mark DeLoura, *Game Programming Gems*, s. 306

3.2.4. Reguła unikania

Reguła unikania (ang. *avoidance*) umożliwia agentowi unikanie przeszkód i przeciwników.

3.3. Implementacja

Ponieważ moja gra bazuje na przestrzeni dyskretnej, musiałem dokonać pewnych zmian w algorytmie, aby dostosować go do swoich potrzeb.

Wszystkie funkcje wyliczające poszczególne reguły są prywatnymi składnikami klasy `Herd`, ponieważ ich wywoływanie poza stadem nie ma sensu.

W algorytmie nie jest w żaden sposób określone, czy bieżący agent ma brać udział w obliczeniach czy nie. Jednak ja skłaniam się ku temu, żeby go w nich nie uwzględniać. Uważam, że pozwoli to na szybsze dostosowywanie się zwierzęcia do zmian otoczenia.

3.3.1. Podstawowe własności klasy `Animal`

```
1.  public abstract class Beeing : Drawable, IUpdateTime {
2.     // ...
3.     public Vector2f ExactPosition {
4.        get {
5.           if ( IsMoveing )
6.              return new Vector2f((float)Lerp(Location.MapPosition.X,
7.                                              GoToField.MapPosition.X, moved_),
8.                                  (float)Lerp(Location.MapPosition.Y,
9.                                              GoToField.MapPosition.Y, moved_));
10.          else
11.             return new Vector2f(Location.MapPosition.X, Location.MapPosition.Y);
12.       }
13.    }
14.
15.    public static float Distance(Beeing from, Beeing to) {
16.       return (float)Sqrt((from.ExactPosition.X - to.ExactPosition.X) *
17.                          (from.ExactPosition.X - to.ExactPosition.X) +
18.                          (from.ExactPosition.Y - to.ExactPosition.Y) *
19.                          (from.ExactPosition.Y - to.ExactPosition.Y));
20.    }
21.    // ...
22. }
```

W klasie `Beeing` właściwość `ExactPosition` określa dokładne położenie na planszy. Różni się ono od własności `Location.MapPosition` tym, że jeśli zwierzę w danym momencie jest w trakcie ruchu, to własność `ExactPosition` będzie odpowiednio przesunięta w stosunku do własności `Location`. W przeciwnym wypadku zostanie zwrócone `Location.MapPosition` zrzutowane na typ `Vector2f`.

Funkcja `Distance` zwraca odległość między dwiema istotami na podstawie ich dokładnego położenia.

```
1.  public class Animal : Beeing {
2.     // ...
3.     public Vector2f MoveVector {
4.        get {
5.           if ( !IsMoveing || (GoToField == null)) { return new Vector2f(0f, 0f); }
6.           Vector2f moveVec = new Vector2f(Location.MapPosition.X - GoToField.MapPosition.X,
7.                                           Location.MapPosition.Y - GoToField.MapPosition.Y);
8.           return moveVec / (float)Length(moveVec);
9.        }
10.    }
11.    // ...
12. }
```

W klasie `Animal` własność `MoveVector`, będąca własnością tylko do odczytu, służy do obliczania kierunku, w jakim porusza się dane zwierzę. Najpierw sprawdzane jest, czy zwierzę w ogóle się porusza. Jeśli tak nie jest, to zwracany jest wektor zerowy. W przeciwnym wypadku obliczana jest różnica między wektorami położenia i celu, która następnie dzielona jest przez długość tego wektora w celu wyznaczenia jednostkowego wektora kierunku.

3.3.2. Reguła rozdzielności

```
1.  private Vector2f Separation(Animal animal, float minDistance) {
2.     return herd_.Where(a => (a != animal) && (Distance(a, animal) < minDistance))
3.                 .Aggregate(new Vector2f(0f, 0f),
4.                            (current, a) =>
5.                               current - new Vector2f(a.ExactPosition.X - animal.ExactPosition.X,
6.                                                      a.ExactPosition.Y - animal.ExactPosition.Y));
7.  }
```

Wybieram ze stada wszystkie zwierzęta poza badanym i sprawdzam kolejno, czy odległość pomiędzy nimi jest mniejsza od minimalnej dozwolonej odległości. Dla tych, dla których powyższy warunek jest spełniony, w zerowym wektorze agreguję różnicę pomiędzy położeniem kolejnych zwierząt a zwierzęciem badanym.

3.3.3. Reguła wyrównania

```
1.  private Vector2f Alignment(Animal animal) {
2.     return herd_.Where(a => a != animal)
3.                 .Aggregate(new Vector2f(0f, 0f),
4.                            (current, a) => current + a.MoveVector);
5.  }
```

Zwykle ta reguła dopasowuje prędkość agenta do pozostałych. Jednak w moim przypadku nie byłoby to całkowicie uzasadnione, ponieważ wartość prędkości poruszania się zwierząt jest stałą. W takim wypadku przyjąłem, że zwierzęta będą dopasowywać jedynie kierunek swojego przemieszczania się.

Wybieram wszystkie zwierzęta poza badanym i w zerowym wektorze agreguję kierunek poruszania się kolejnych zwierząt.

3.3.4. Reguła spójności

```
1.  private Vector2f Cohesion(Animal animal) {
2.     Vector2f centerOfMass = herd_.Where(a => a != animal)
3.                                  .Aggregate(new Vector2f(0f, 0f),
4.                                             (current, a) =>
5.                         current + new Vector2f(a.ExactPosition.X, a.ExactPosition.Y));
6.
7.     centerOfMass /= herd_.Count - 1;
8.
9.     Vector2f animalPos = new Vector2f(animal.ExactPosition.X, animal.ExactPosition.Y);
10.
11.    return centerOfMass - animalPos;
12. }
```

Wybieram ze stada wszystkie zwierzęta poza badanym i w zerowym wektorze agreguję położenia kolejnych zwierząt. Tak otrzymaną sumę dzielę przez liczbę zwierząt -1, dlatego że badane zwierzę nie zostało uwzględnione w obliczeniach, z powodu wspomnianego na początku rozdziału.

3.3.5. ZARZĄDZANIE STADEM

```
1.  public void UpdateTime(object sender, UpdateEventArgs e) {
2.     if ( counter_ == 0 ) {
3.        counter_ = rand_.Next(1000, 3000);
4.
5.        if ( rand_.Next(2) == 1 ) {
6.           pocX_ = 0;
7.           konX_ = 5;
8.        } else {
9.           pocX_ = -5;
10.          konX_ = 0;
11.       }
12.
13.       if ( rand_.Next(2) == 1 ) {
14.          pocY_ = 0;
15.          konY_ = 5;
16.       } else {
17.          pocY_ = -5;
18.          konY_ = 0;
19.       }
20.    }
21.
22.    // ALGORYTM STADA
23.    foreach ( Animal animal in herd_.Where(animal => !animal.IsMoveing) ) {
24.       // wyliczenie pozycji na jaka powinno poruzyc sie zwierze
25.       Vector2f randVec = new Vector2f(rand_.Next(pocX_, konX_), rand_.Next(pocY_, konY_));
26.       Vector2f moveVector = Cohesion(animal) +
27.                             Separation(animal, 2f) +
28.                             Alignment(animal) +
29.                             randVec;
30.
31.       MapField field;
32.
33.       // wylicenie jakie to pole na mapie
34.       int x = (int)Round(moveVector.X);
35.       int y = (int)Round(moveVector.Y);
36.       try { field = animal.Location.Neighbour[x, y]; }
37.       catch ( NoSouchNeighbourException ) { continue; }
38.
39.       // przekazanie tego pola do zwierzecia
40.       animal.GoToField = field;
41.    }
42.
43.    --counter_;
44. }
```

W każdym cyklu na samym początku sprawdzany jest licznik, odpowiadający liczbie cykli, podczas których dane stado ma się poruszać w konkretnym kierunku. Jeśli jest on równy zero, to losowana jest nowa jego wartość oraz nowy kierunek, który ustalany jest na podstawie dwóch pseudolosowych wartości. Na samym końcu każdego uaktualnienia wartość licznika jest dekrementowana.

Moja plansza składa się z dyskretnej siatki, dlatego nie jest możliwa zmiana kierunku poruszania się, ponieważ obiekty mogą z danego pola przemieścić się na co najwyżej ośmiu jego sąsiadów. Tak więc przy każdym uaktualnieniu uruchamiany jest algorytm stada tylko dla tych zwierząt, które się nie ruszają. Poza opisanymi wyżej trzema regułami do nowego wektora, mającego wyznaczyć cel zwierzęciu, dodawany jest też wektor, którego składowe są pseudolosowymi liczbami. Służy on temu, żeby stado powoli zmierzało w jakimś kierunku. Ponadto w początkowej implementacji bez pseudolosowego wektora zdarzyło mi się kilkukrotnie zaobserwować, że stado osiąga stan równowagi. Wtedy wszystkie reguły zwracają takie wartości, które się nawzajem znoszą bądź końcowy wektor ma tak niewielkie składowe, że po zaokrągleniu składowe wektora `moveVector` wynoszą

cały czas zero. Następnie składowe wektora są zaokrąglane. Dzieje się tak dlatego, że plansza składa się z pól o całkowitoliczbowych współrzędnych. Na koniec sprawdzane jest po pierwsze, czy wyznaczone pole w ogóle istnieje, a po drugie, czy można się na nie przemieścić. Jeśli którykolwiek z powyższych warunków nie jest spełniony, to zwierzę nie dostanie w tym momencie instrukcji i poczeka do następnego cyklu. W przeciwnym wypadku zwierzę dostanie instrukcje, na jakie pole ma się kierować.

4. LOGIKA ROZMYTA

Logika klasyczna opiera się na dwóch wartościach: *prawdzie* i *fałszu*. Są one często reprezentowane odpowiednio przez cyfry 1 i 0. Granica pomiędzy tymi stanami jest jednoznacznie zdefiniowana i nie istnieją żadne stany pośrednie. Jest to bardzo proste z punktu widzenia implementacji, jednak równocześnie bardzo dalekie od tego, w jaki sposób człowiek postrzega świat. Posiadamy bowiem bardzo rozbudowany system niejasnych reguł podziału. Coś może być „bardziej" bądź „mniej" jakieś, może być „trochę" lub „prawie" czymś.

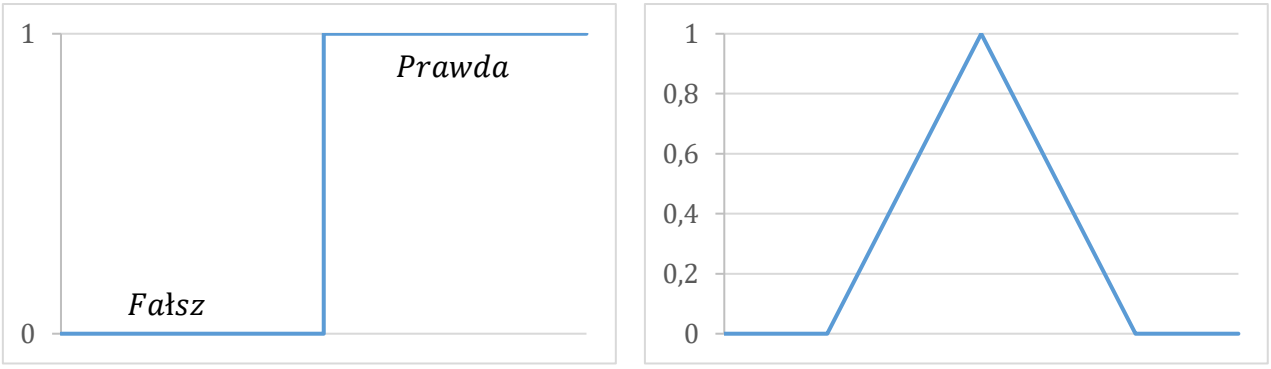
4.1. PODSTAWOWA RÓŻNICA WZGLĘDEM LOGIKI KLASYCZNEJ

Taki sposób komunikacji z komputerem umożliwia nam logika rozmyta (ang. *fuzzy logic*) stworzona przez Lotfi Zadeha w latach 60.[22] Dzięki niej komputer jest w stanie zrozumieć takie rozmyte zasady.

W przeciwieństwie do logiki klasycznej, którą w tym wypadku dobrze opisuje słowo *ostra*, zmienna w logice rozmytej może przyjąć dowolną wartość z przedziału $[0, 1]$, którą można przetłumaczyć jako na przykład „prawie prawda", „w połowie fałsz".

4.2. ZBIÓR ROZMYTY I ROZMYTA ZMIENNA LINGWISTYCZNA

W logice klasycznej musimy orzec, czy wartość przynależy do danego zbioru. Natomiast w logice rozmytej dodatkowo musimy określić stopień tej przynależności. Może to doprowadzić do sytuacji, w której dany obiekt przynależałby do rozłącznych zbiorów z punktu widzenia logiki klasycznej. Po prostu będzie on miał w tych zbiorach różny stopień przynależności. Na przykład może być 0,5 zdrowy i 0,5 ranny.



Rysunek 4.1. Wykresy obrazują różnice pomiędzy wartościami w logice klasycznej (po lewej) i w logice rozmytej (po prawej)

W ujęciu czysto matematycznym zbiorem rozmytym A w przestrzeni $\mathbb{X}$ jest zbiór uporządkowanych par:

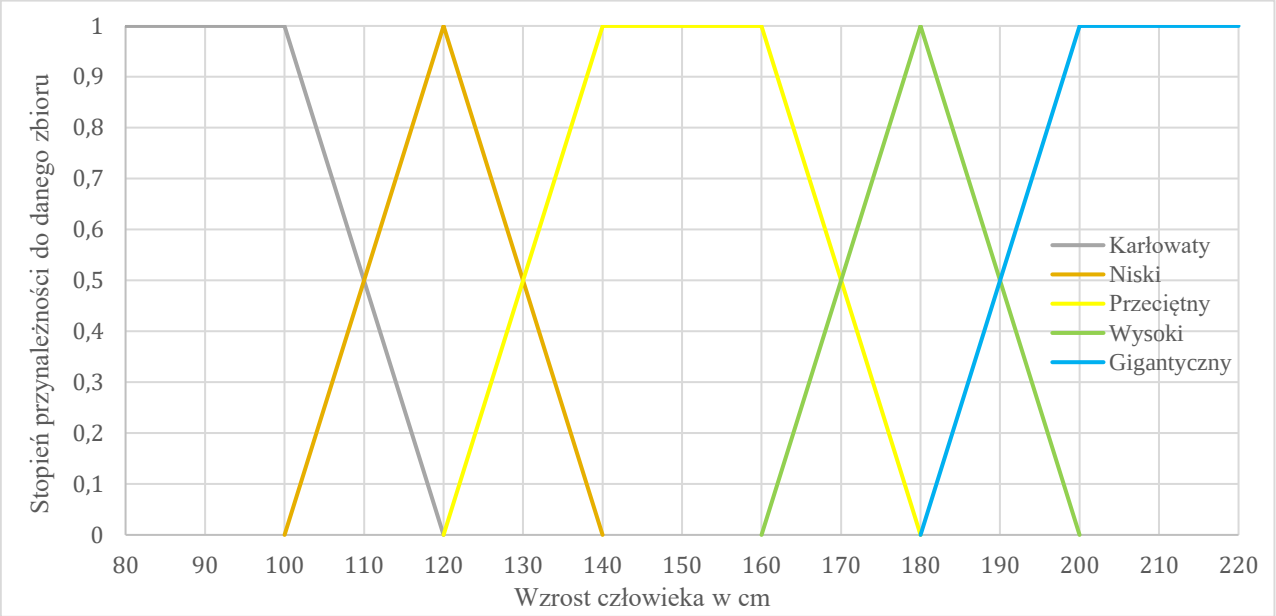
$$A = \{(x, \mu_A(x)) \mid x \in \mathbb{X}\},$$

gdzie $\mu_A: X \to [0,1]$ nazywamy funkcją przynależności zmiennej x do zbioru A[23].

Zbiory rozmyte określające pewien wspólny koncept można łączyć w rozmyte zmienne lingwistyczne (ang. *fuzzy linguistic variables*, *FLV*)[24]. Przykładem takiej zmiennej lingwistycznej może

[22] Mat Buckland, *Programming Game AI by Example*, s. 416
[23] *Fuzzy set* [https://en.wikipedia.org/wiki/Fuzzy_set]
[24] Mat Buckland, *Programming Game AI by Example*, s. 423-424

być na przykład $wzrost\ człowieka$, na który składają się takie zbiory rozmyte jak: $Karłowaty$, $Niski$, $Przeciętny$, $Wysoki$ i $Gigantyczny$.

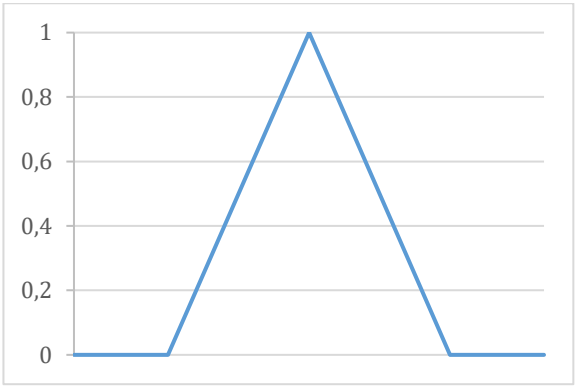


Wykres 4.1. Reprezentacja graficzna rozmytego zbioru wzrostu ludzi

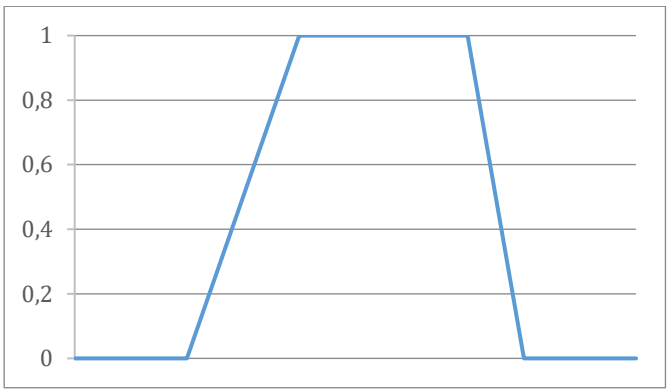
4.3. FUNKCJA PRZYNALEŻNOŚCI DO ZBIORU

Funkcja przynależności reprezentuje sposób, w jaki zmienne wejściowe przekładane są na stopień przynależności do danego zbioru[25]. Może ona przybrać dowolny kształt, ale zazwyczaj jest triangularna bądź trapezoidalna[26].

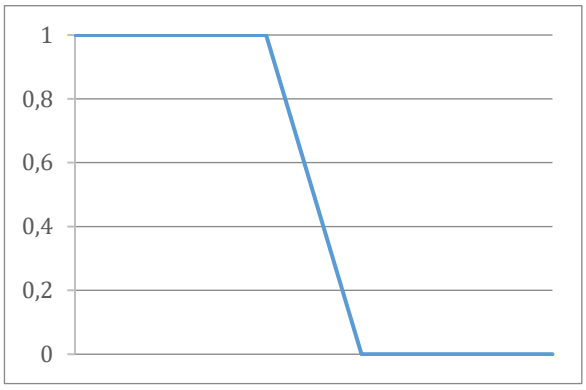
[25] David M. Bourg, Glenn Seeman, *AI for Game Developers*, s. 193
[26] Ian Millington, *Artificial intelligence for games*, s. 419



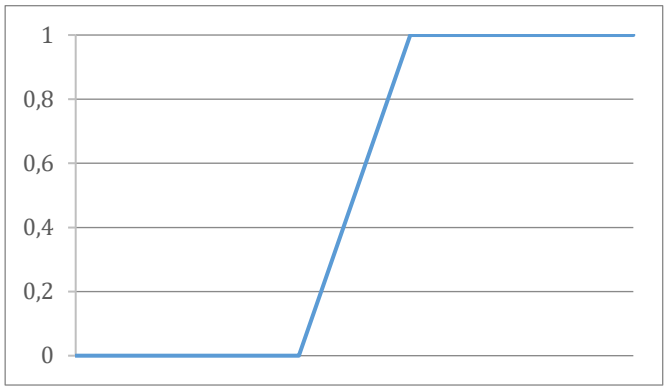
1. Funkcja triangularna



2. Funkcja trapezoidalna



3. Funkcja typu left shoulder



4. Funkcja typu right shoulder

Rysunek 4.2. Cztery najczęściej stosowane funkcje przynależności

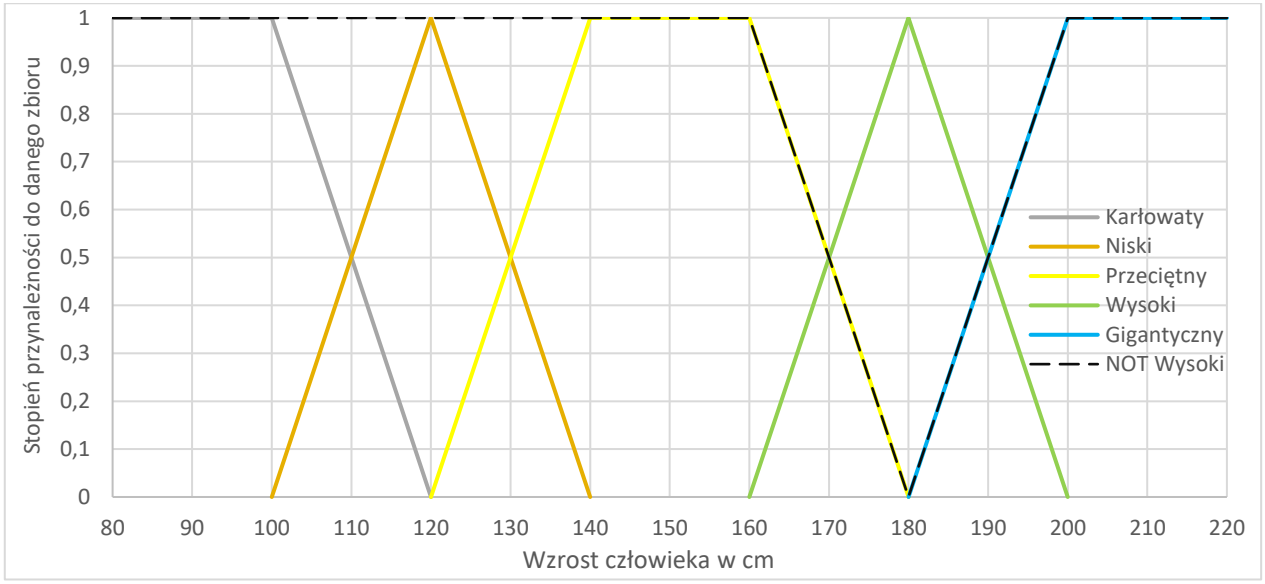
4.4. OPERACJE NA ZBIORACH ROZMYTYCH

Tak jak w wypadku logiki klasycznej, również w logice rozmytej występują operatory logiczne takie jak AND, OR, NOT i inne. Są one najczęściej definiowane następująco[27]:

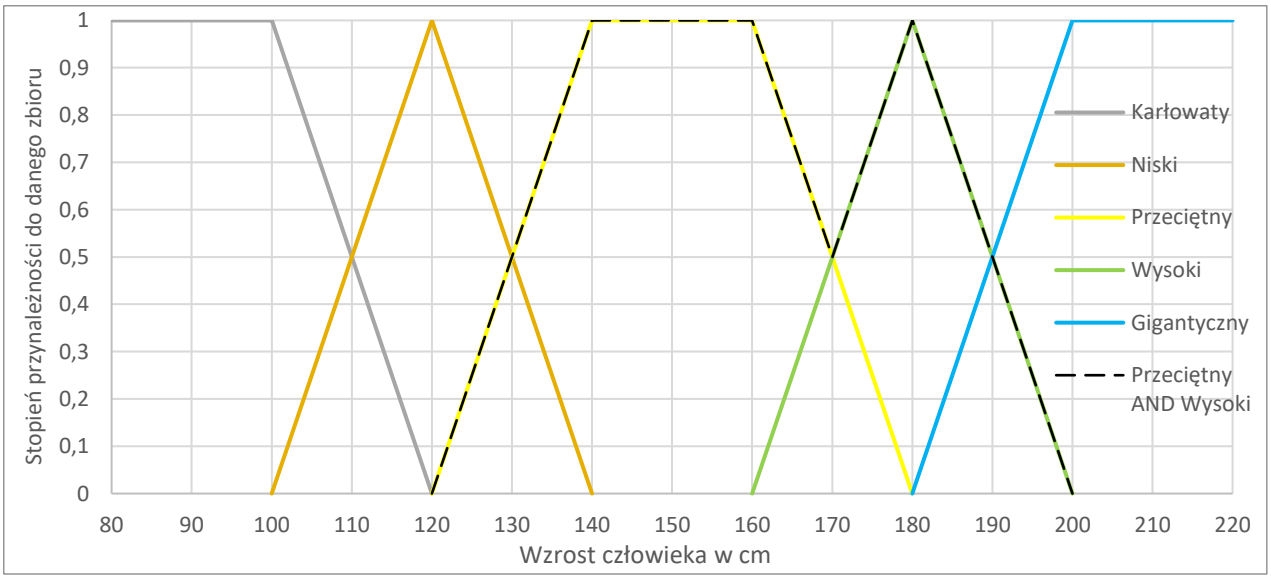
Wyrażenie	Równoważne	Równanie
NOT A		$1 - \mu_A$
A AND B		$\min(\mu_A, \mu_B)$
A OR B		$\max(\mu_A, \mu_B)$
A XOR B	NOT(B) AND A	$\min(\mu_A, 1 - \mu_B)$
	NOT(A) AND B	$\min(1 - \mu_A, \mu_B)$
A NOR B	NOT(A OR B)	$1 - \max(\mu_A, \mu_B)$
A NAND B	NOT(A AND B)	$1 - \min(\mu_A, \mu_B)$

gdzie A i B to zbiory rozmyte, a μ_A i μ_B to funkcje przynależności do nich.

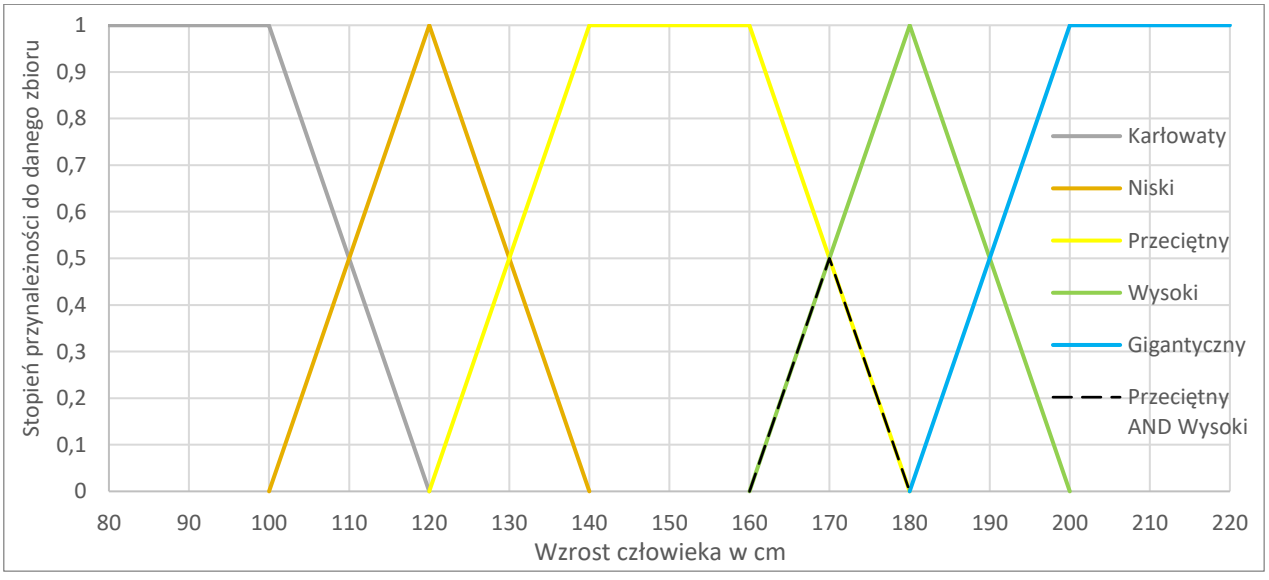
[27] Por. Ian Millington, *Artificial intelligence for games*, s. 353; David M. Bourg, Glenn Seeman, *AI for Game Developers*, s. 200-201; Mat Buckland, *Programming Game AI by Example*, s. 421-423



Wykres 4.2. Reprezentacja graficzna rozmytego zbioru wzrostu ludzi.
Linią przerywaną zaznaczona operacja NOT Wysoki.



Wykres 4.3. Reprezentacja graficzna rozmytego zbioru wzrostu ludzi.
Linią przerywaną zaznaczona operacja Przeciętny OR Wysoki.



Wykres 4.4. Reprezentacja graficzna rozmytego zbioru wzrostu ludzi.
Linią przerywaną zaznaczona operacja Przeciętny AND Wysoki.

Jednak tylko operator NOT ma jedną definicję. W pozostałych przypadkach funkcjonują inne definicje operatorów AND i OR (a co za tym idzie także pozostałych operatorów)[28]. Dla sumy logicznej AND wzory te zwane są operatorami s-normy, a dla iloczynu logicznego OR – t-normy.

Nazwa operatora	Wzór
Minimum	$\mu_{A\cap B}(x) = \min\left(\mu_A(x), \mu_B(x)\right)$
Iloczyn	$\mu_{A\cap B}(x) = \mu_A\mu_B$
Iloczyn Hamachera	$\mu_{A\cap B}(x) = \frac{\mu_A\mu_B}{\mu_A + \mu_B - \mu_A\mu_B}$
Iloczyn Einsteina	$\mu_{A\cap B}(x) = \frac{\mu_A\mu_B}{2 - (\mu_A + \mu_B - \mu_A\mu_B)}$
Iloczyn drastyczny	$\mu_{A\cap B}(x) = \begin{cases}\min\left(\mu_A(x), \mu_B(x)\right), & dla\ \max\left(\mu_A(x), \mu_B(x)\right) = 1 \\ 0, & w\ pozostałych\ przypadkach\end{cases}$
Ograniczona różnica	$\mu_{A\cap B}(x) = \max(0, \mu_A(x) + \mu_B(x) - 1)$

Tabela 4.1.[29] Operatory t-normy

Nazwa operatora	Wzór
Minimum	$\mu_{A\cup B}(x) = \max\left(\mu_A(x), \mu_B(x)\right)$
Suma algebraiczna	$\mu_{A\cup B}(x) = \mu_A + \mu_B - \mu_A\mu_B$
Suma Hamachera	$\mu_{A\cup B}(x) = \frac{\mu_A + \mu_B - 2\mu_A\mu_B}{1 - \mu_A\mu_B}$
Suma Einsteina	$\mu_{A\cup B}(x) = \frac{\mu_A + \mu_B}{1 + \mu_A\mu_B}$
Suma drastyczna	$\mu_{A\cup B}(x) = \begin{cases}\max\left(\mu_A(x), \mu_B(x)\right), & dla\ \min\left(\mu_A(x), \mu_B(x)\right) = 0 \\ 1, & w\ pozostałych\ przypadkach\end{cases}$
Ograniczona suma	$\mu_{A\cup B}(x) = \min\left(1, \mu_A(x) + \mu_B(x)\right)$

Tabela 4.2.[29] Operatory s-normy

4.5. ALGORYTM WNIOSKOWANIA ROZMYTEGO

Wnioskowanie rozmyte oparte jest na stopniach przynależności do zbiorów rozmytych. Ponieważ nie jest to sposób, w jaki zwykle przechowuje się dane w grach, dlatego też potrzebny jest pewien sposób konwersji na zmienne rozmyte (ang. *fuzzy values*). Proces ten nazywa się rozmyciem zmiennych (fuzyfikacja, ang. *fuzzification*). Następnie przeprowadzany jest potrzebny proces wnioskowania rozmytego, na wyjściu którego otrzymywany jest rozmyty zbiór wartości z ich stopniem przynależności. Aby móc z niego nadal korzystać, trzeba zamienić dane z powrotem na pewną wartość ostrą (ang. *crisp value*). Proces ten nazywa się wyostrzaniem (defuzyfikacja, ang. *defuzzification*).

4.5.1. ROZMYCIE (FUZYFIKACJA)

Najpopularniejszą techniką rozmycia jest zmiana pewnej zmiennej liczbowej, jak na przykład wzrostu gracza, na stopień przynależności do jednego lub więcej zbiorów rozmytych. Aby to osiągnąć, wystarczy odpowiednio przygotować funkcję przynależności, a ona dla naszej danej zwróci stopień przynależności do danego zbioru.

[28] Por. Ian Millington, *Artificial intelligence for games*, s. 353; David M. Bourg, Glenn Seeman, *AI for Game Developers*, s. 201
[29] Jan Kalwoda, *Podstawy logiki rozmytej* [http://www.isep.pw.edu.pl/ZakladNapedu/dyplomy/fuzzy/podstawy_FL.htm]

Na przykład dla człowieka o wzroście $175\ cm$ proces fuzyfikacji będzie przebiegał następująco:

1. $Karłowaty(175) = 0$
2. $Niski(175) = 0$
3. $Przeciętny(175) = 0{,}25$
4. $Wysoki(175) = 0{,}75$
5. $Gigantyczny(175) = 0$

W tym przypadku wartości po rozmyciu sumują się do jedności, jednak nie ma takiego wymogu. Ponadto dla jednej wartości ostrej nie ma żadnych ograniczeń co do liczby funkcji rozmywających, jakich można na niej użyć w danym momencie[30].

W przypadku danych takich jak wartości `bool` czy `enum`, częstym jest przechowywanie predefiniowanych stopni przynależności do danych zbiorów rozmytych. Wtedy proces fuzyfikacji sprowadza się do przekazania dalej tych wartości[31].

4.5.2. WNIOSKOWANIE ROZMYTE

Wnioskowanie rozmyte na podstawie danych zmiennych, dla których znane są stopnie przynależności do pewnych zbiorów rozmytych, tworzy inne zmienne rozmyte należące do innych zbiorów.

Sprowadza się to zwykle do pewnych reguł[32], które składają się z przyczyny i skutku. Ich ogólna forma jest następująca:

JEŚLI $przyczyna$ WTEDY $skutek$.

Widać, że takie reguły nie różnią się za bardzo od tych, które programiści implementują, gdy używają logiki klasycznej. Jednak samo ich działanie różni się znacząco. W normalnym wypadku otrzymuje się w wyniku odpowiedź, czy skutek ma zajść, czy nie. W wypadku wnioskowania rozmytego w odpowiedzi dostarczana jest wartość, w jakim stopniu skutek ma zajść.

Przykładowe reguły rozmyte:

- JEŚLI $\{DużoZdrowia \rightarrow 0{,}7\}$ AND $\{MałoLeniwy \rightarrow 0{,}6\}$ WTEDY $\{Pracuj \rightarrow 0{,}68\}$;
- JEŚLI $\{Przeciwnik.MałoZdrowia \rightarrow 0{,}95\}$ WTEDY $\{AtakujAgresywnie \rightarrow 0{,}9\}$;
- JEŚLI $\{Przeciwnik.Niedaleko \rightarrow 0{,}84\}$ AND $\{DużoAmunicji \rightarrow 0{,}75\}$ WTEDY $\{Atakuj \rightarrow 0{,}79\}$.

4.5.3. WYOSTRZENIE (DEFUZYFIKACJA)

Istnieje kilka metod wyostrzenia zmiennych i oczywiście nie można wśród nich wskazać tej najlepszej. Mają one podobną strukturę, jednak różnią się wydajnością i stabilnością.

4.5.3.1. NAJWYŻSZY STOPIEŃ PRZYNALEŻNOŚCI

Najprostszym sposobem jest po prostu wybranie zmiennej, która ma najwyższy stopień przynależności do swojego zbioru.

W przypadku, gdy w taki sposób otrzymamy wartość 1, zwykle wybierany jest jeden z czterech punktów[33]:

[30] Ian Millington, *Artificial intelligence for games*, s. 346
[31] Ian Millington, *Artificial intelligence for games*, s. 346-347
[32] Mat Buckland, *Programming Game AI by Example*, s. 424-425
[33] Ian Millington, *Artificial intelligence for games*, s. 348

- minimum maksima – obliczane jako najmniejsza wartość, dla której funkcja przynależności zwróci wartość 1;
- maksimum maksima – obliczane jako największa wartość, dla której funkcja przynależności zwróci wartość 1;
- średnia wartość maksima – obliczana jako średnia dwóch powyższych;
- wartość środkowa – obliczana poprzez przecałkowanie powierzchni pod krzywą funkcji przynależności i znalezienie w ten sposób punktu na krzywej dzielącego ten obszar na dwie równe części.

Ponieważ proces całkowania może zajmować dużo czasu procesora, zwykle przeprowadzany jest raz, najlepiej znacznie wcześniej niż wynik ten będzie wymagany.

Główną zaletą tego sposobu jest jego prostota i szybkość działania. Jednak sporą wadą jest to, że dla różnych danych może zwracać taki sam wynik.

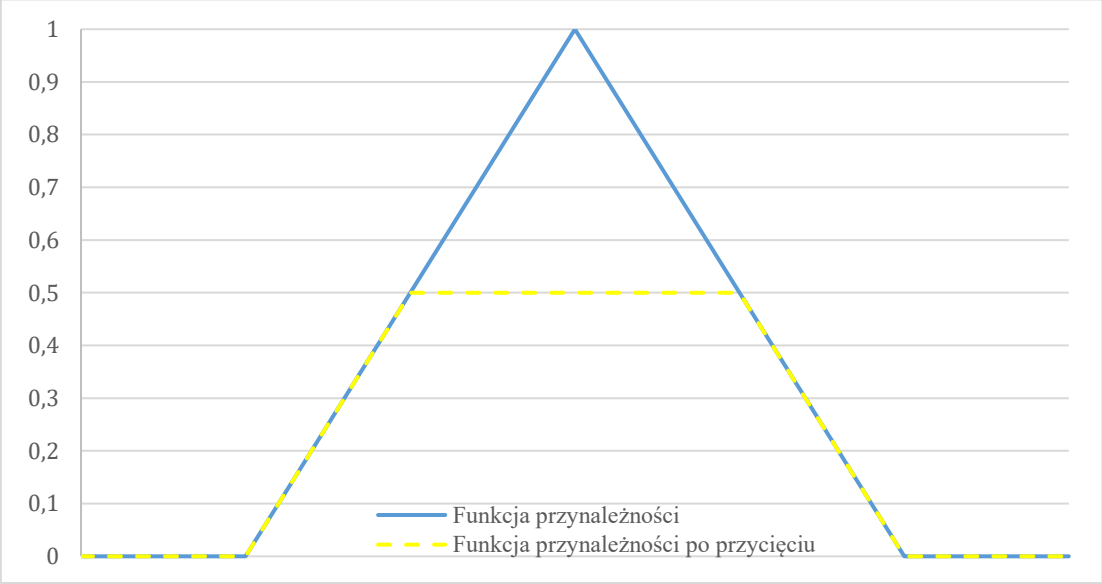
4.5.3.2. ŁĄCZENIE W OPARCIU O FUNKCJE PRZYNALEŻNOŚCI

Łatwym sposobem na obejście tego problemu jest zmieszanie tych wartości na podstawie ich stopnia przynależności do danych zbiorów. W przypadku, gdy wartość jest równa 1, należy użyć jednej z wyżej opisanych metod. Przy takim łączeniu ważne jest, aby wartości stopnia przynależności były znormalizowane, ponieważ inaczej można by uzyskać bezsensowne dane, na przykład większe od dopuszczalnego maksimum.

4.5.3.3. ŚRODEK CIĘŻKOŚCI

Metoda z wyznaczaniem środka ciężkości (ang. *centroid*, *centre of mass*) jest najbardziej dokładna, ale też najbardziej złożona[34].

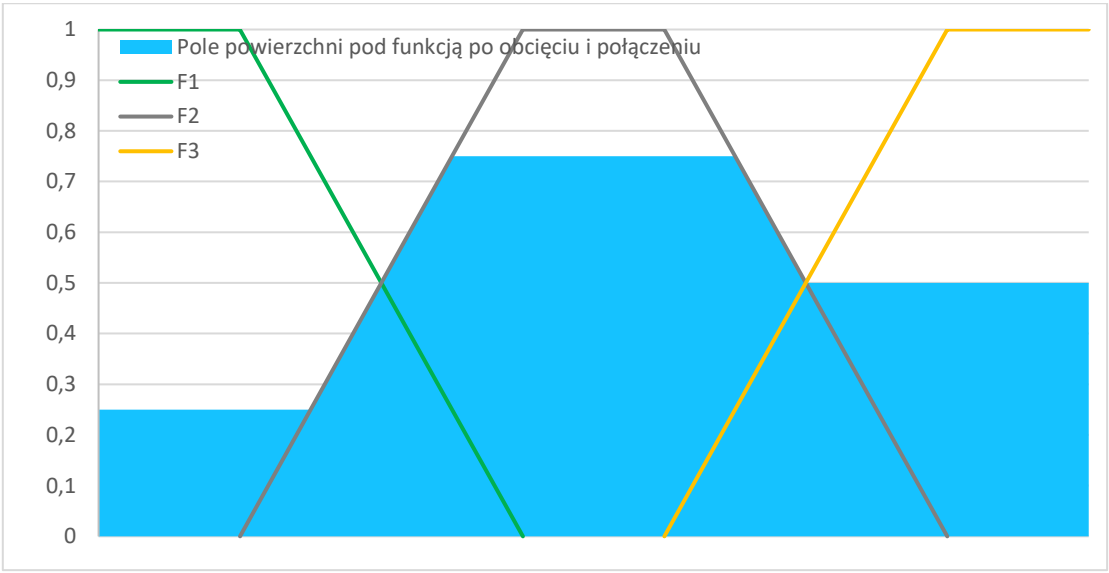
Na początku każda funkcja przynależności zostaje ścięta do najwyższej wartości z odpowiadającego jej zbioru. Na przykład jeśli po wnioskowaniu najwyższą wartością w zbiorze jest przynależność na poziomie 0,5, to funkcja odpowiadająca temu zbiorowi zostanie do tej wartości przycięta.



Wykres 4.5. Pierwotna funkcja przynależności oraz funkcja po przycięciu

Tak otrzymane funkcje są łączone w jedną, której pole powierzchni jest następnie całkowane, żeby znaleźć punkt środkowy.

[34] Mat Buckland, *Programming Game AI by Example*, s. 434



Wykres 4.6. Funkcje przynależności oraz pole powierzchni pod funkcją powstałą przez odpowiednie obcięcie i połączenie poszczególnych funkcji.
Obcięcia dokonano na poziomie: 0,25 *dla F1,* 0,75 *dla F2 oraz* 0,5 *dla F3.*

Niestety w tym wypadku nie można całkowania wykonać wcześniej i zachować wyniku, ponieważ nie znamy wartości, na jakiej funkcje przynależności będą obcięte.

4.6. Implementacja

Na potrzeby gry napisałem prosty silnik logiki rozmytej. Kieruje on wyborem zadania, które ma wykonać kolonista oraz zachowaniem atakujących przeciwników.

4.6.1. Klasa zmiennej rozmytej

```
1.  public abstract class FuzzyVariable {
2.     public FuzzyVariable(float value) { Value = value; }
3.
4.     public virtual float Value { get; set; }
5.
6.     public float FuzzyValue {
7.        get {
8.           if ( State == null ) throw new VariableNotFuzzifiedException();
9.
10.          State = null;
11.          return fuzzyVar_;
12.       }
13.       protected set { fuzzyVar_ = value; }
14.    }
15.
16.    public string State { get; protected set; }
17.
18.    public abstract int StatesCount { get; }
19.
20.    public abstract void Fuzzify(string state);
21.
22.    public abstract void Fuzzify(int stateNo);
23.
24.    public static implicit operator float(FuzzyVariable fuzzy) => fuzzy.Value;
25.
26.    private float fuzzyVar_;
27.
28.    public static float And(float first, float second) => Min(first, second);
29.
30.    public static float Or(float first, float second) => Max(first, second);
31.
32.    public static float Not(float val) => 1f - val;
33. }
```

Abstrakcyjna klasa `FuzzyVariable` jest podstawową klasą w mojej implementacji logiki rozmytej. Odpowiada ona rozmytej zmiennej lingwistycznej. Konstruktor inicjalizuje jej instancję zadaną wartością. Właściwość `Value` przechowuje ostrą zmienną (*crisp value*). Jest ona wirtualna, ponieważ typy dziedziczące po `FuzzyVariable` mogą potrzebować na przykład pewnych ograniczeń dla wartości zmiennej. Natomiast właściwość `FuzzyValue` przechowuje stopień przynależności zmiennej rozmytej (*fuzzy value*) do zbioru, dla jakiego rozmyto daną zmienną. Własność `State` przechowuje nazwę zbioru, dla jakiej w obecnym stanie rozmyta jest zmienna. Jeśli jest równa `null`, to znaczy, że zmienna nie została rozmyta. W tym wypadku próba odczytania wartości `FuzzyValue` spowoduje rzucenie wyjątku. Jest to zabezpieczenie przed odczytywaniem nierozmytej zmiennej. Własność `StatesCount` zwraca informację, ile zbiorów rozmytych składa się na daną zmienną. Dwie przeciążone metody `Fuzzify` odpowiadają za rozmycie zmiennej. Ze względu na fakt, że zmienne rozmyte fuzyfikuję w pętli, potrzebowałem wygodnego sposobu przekładania numeru iteracji na nazwę zbioru, dla którego zmienna ma zostać rozmyta – do tego właśnie służy wersja z argumentem typu `int`. Wywołuje ona na podstawie otrzymanej liczby funkcję z argumentem typu `string`, która z kolei inicjalizuje odpowiednio własności `State` i `FuzzyValue`. Ponadto, żeby umożliwić wygodne korzystanie ze zmiennych ostrych, które mogą być wykorzystywane w kodzie oraz żeby każdorazowe odwoływanie się do własności `Value` nie było koniecznie, zaimplementowałem niejawną konwersję typu `FuzzyVariable` na typ `float`.

W tej klasie znajdują się też statyczne metody `And`, `Or` oraz `Not`, które odpowiadają za obliczanie wyników operacji logicznych na stopniach przynależności do zbioru. Zaimplementowałem je zgodnie z najprostszym z opisanych w rozdziale 4.4. sposobem, głównie z powodu jego prostoty i wysokiej wydajności.

4.6.2. Funkcje przynależności do zbioru rozmytego

Zaimplementowałem cztery funkcje przynależności do zbioru rozmytego. Wszystkie korzystają ze stworzonej przeze mnie funkcji wyliczającej współczynniki równania liniowego. Sposób jej działania oparłem na własnym rozwiązaniu bardzo prostego układu dwóch równań liniowych.

$$\begin{cases} y_1 = ax_1 + b & (1) \\ y_2 = ax_2 + b & (2) \end{cases}$$

Następnie należy od równania (1) odjąć równanie (2), co daje:

$$y_1 - y_2 = a(x_1 - x_2)$$
$$a = \frac{y_1 - y_2}{x_1 - x_2}.$$

Znając wartość współczynnika a, możemy podstawić go do jednego z równań początkowych i wyliczyć wartość współczynnika b:

$$y_1 = ax_1 + b$$
$$b = y_1 - ax_1.$$

```
1. public static (double a, double b) LinearFactors(double x1, double y1, double x2, double y2) {
2.    double a = (y1 - y2) / (x1 - x2);
3.    double b = y1 - a * x1
4.
5.    return (a, b);
6. }
```

Funkcja `LinearFactors` zwraca krotkę składającą się z wyliczonych współczynników a i b prostej przechodzącej przez punkty o współrzędnych $(x1, y2)$ i $(x2, y2)$.

Dla przejrzystości kodu funkcje te zaimplementowałem jako osobne struktury. Niestety język C# nie pozwala na przeładowanie operatora `()`, więc zmuszony byłem wywołanie funkcji zrealizować metodą `Invoke`.

4.6.2.1. FUNKCJE LEFT I RIGHT SHOULDER

```
1.  public struct LeftShoulderFunc {
2.     public LeftShoulderFunc(double x0, double x1) {
3.        x0_ = x0;
4.        x1_ = x1;
5.        (a_, b_) = LinearFactors(x0, 0, x1, 1);
6.     }
7.
8.     public double Invoke(double x) {
9.        if ( x <= x0_ ) return 0.0;
10.       else if ( x >= x1_ ) return 1.0;
11.       else return a_ * x + b_;
12.    }
13.
14.    private readonly double x0_, x1_, a_, b_;
15. }
```

```
1.  public struct RightShoulderFunc {
2.     public RightShoulderFunc(double x0, double x1) {
3.        x0_ = x0;
4.        x1_ = x1;
5.        (a_, b_) = LinearFactors(x0, 1, x1, 0);
6.     }
7.
8.     public double Invoke(double x) {
9.        if ( x <= x0_ ) return 1.0;
10.       else if ( x >= x1_ ) return 0.0;
11.       else return a_ * x + b_;
12.    }
13.
14.    private readonly double x0_, x1_, a_, b_;
15. }
```

Funkcja `LeftShoulderFunc.Invoke` ma przebieg zgodny z wykresem 3. z rysunku 4.2. natomiast `RightShoulderFunc.Invoke` ma przebieg zgodny z wykresem 4. z tego samego rysunku.

4.6.2.2. FUNKCJA TRIANGULARNA

```
1.  public struct TraingularFunc {
2.     public TraingularFunc(double x0, double x1, double x2) {
3.        x0_ = x0;
4.        x1_ = x1;
5.        x2_ = x2;
6.        (a1_, b1_) = LinearFactors(x0, 0, x1, 1);
7.        (a2_, b2_) = LinearFactors(x1, 1, x2, 0);
8.     }
9.
10.    public double Invoke(double x) {
11.       if ( x <= x0_ || x >= x2_ ) return 0.0;
12.       else if ( x < x1_ ) return a1_ * x + b1_;
13.       else return a2_ * x + b2_;
14.    }
15.
16.    private readonly double x0_, x1_, x2_, a1_, b1_, a2_, b2_;
17. }
```

Funkcja `TraingularFunc.Invoke` ma przebieg zgodny z wykresem 1. z rysunku 4.2.

4.6.2.3. FUNKCJA TRAPEZOIDALNA

```
1.  public struct TrapezoidFunc {
2.     public TrapezoidFunc(double x0, double x1, double x2, double x3) {
3.        x0_ = x0;
4.        x1_ = x1;
5.        x2_ = x2;
6.        x3_ = x3;
7.
8.        (a1_, b1_) = LinearFactors(x0, 0, x1, 1);
9.        (a2_, b2_) = LinearFactors(x2, 1, x3, 0);
10.    }
11.
12.    public double Invoke(double x) {
13.       if ( x <= x0_ || x >= x3_ ) return 0.0;
14.       else if ( x1_ < x && x < x2_ ) return 1.0;
15.       else if ( x0_ < x && x < x1_ ) return a1_ * x + b1_;
16.       else return a2_ * x + b2_;
17.    }
18.
19.    private readonly double x0_, x1_, x2_, x3_, a1_, b1_, a2_, b2_;
20. }
```

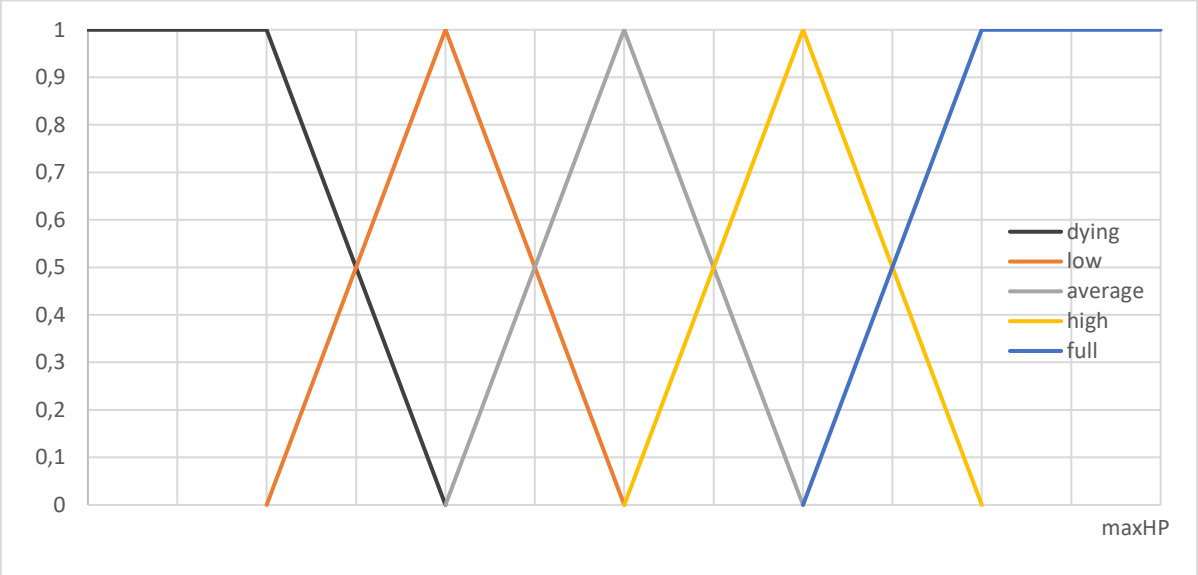
Funkcja `TrapezoidFunc.Invoke` ma przebieg zgodny z wykresem 2. z rysunku 4.2.

4.6.3. ZAIMPLEMENTOWANE ZMIENNE ROZMYTE

Zaimplementowałem cztery zmienne rozmyte. Wszystkie dziedziczą po klasie `FuzzyVariable`.

4.6.3.1. FuzzyHP

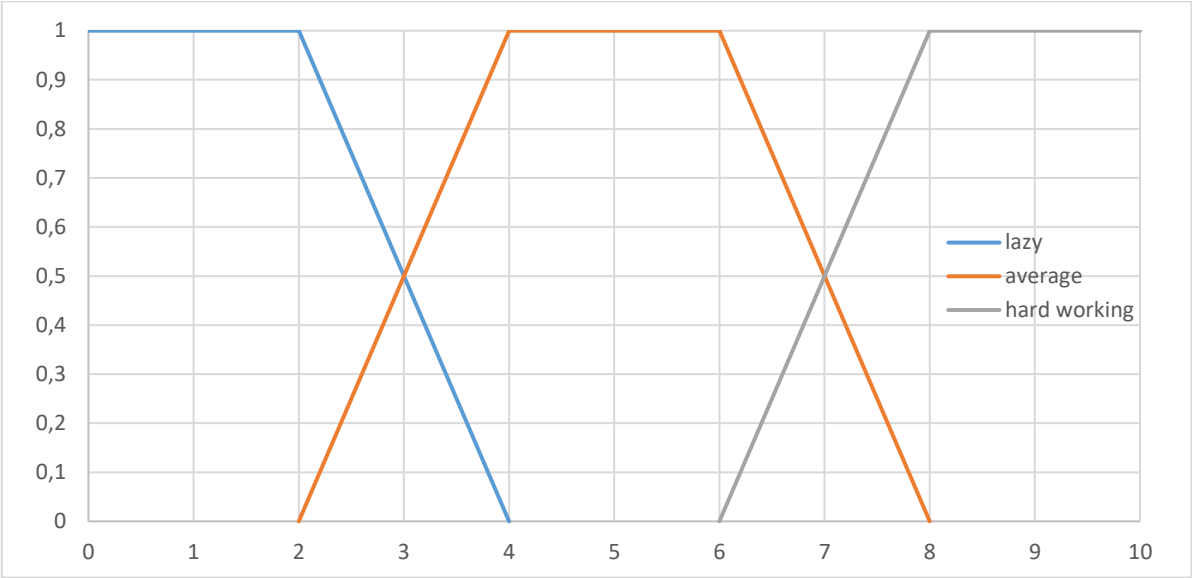
Zmienna `FuzzyHP` reprezentuje poziom życia istoty. Posiada ona dodatkową własność `MaxHP`, która przechowuje maksymalną żywotność dla tej istoty. Jeśli nastąpi próba przypisania wartości większej, zostanie przypisana wartość `MaxHP`. Jeśli zmieni się wartość `MaxHP`, wszystkie funkcje zostaną zdefiniowane na nowo i dopasowane do nowej wartości.



Wykres 4.7. Reprezentacja graficzna zbiorów rozmytych zawartych w zmiennej `FuzzyHP`

4.6.3.2. `FuzzyLaziness`

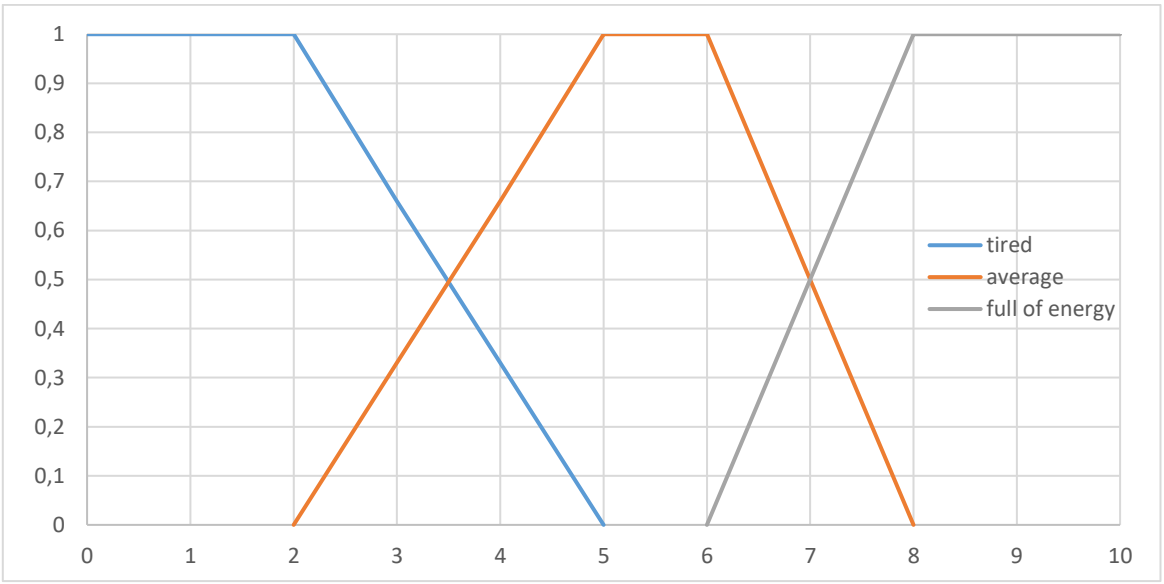
Zmienna `FuzzyLaziness` reprezentuje poziom lenistwa postaci.



Wykres 4.8. Reprezentacja graficzna zbiorów rozmytych zawartych w zmiennej `FuzzyLaziness`

4.6.3.3. `FuzzyRest`

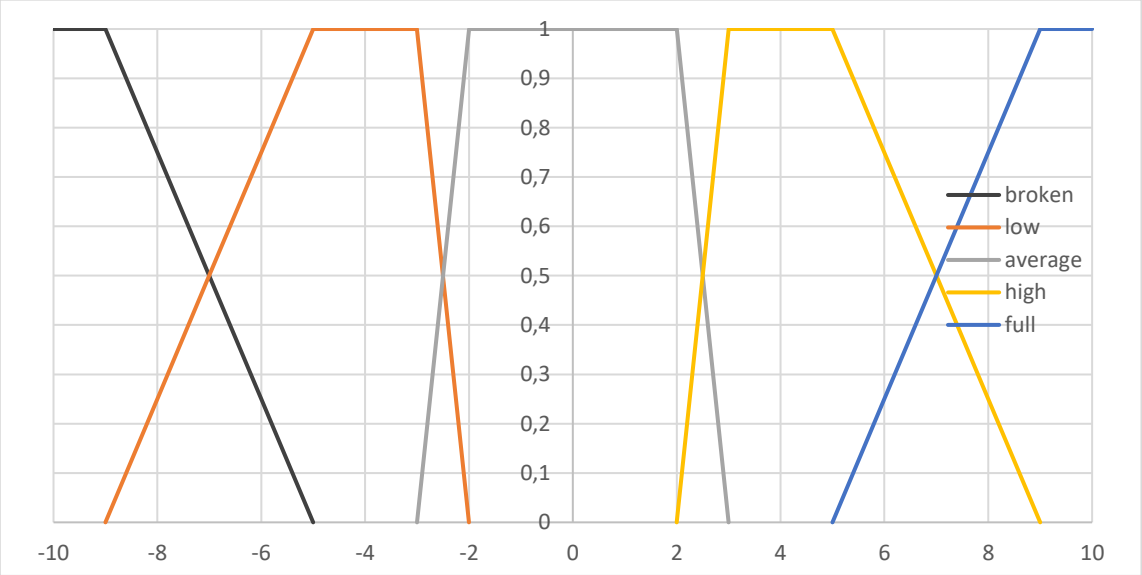
Zmienna `FuzzyRest` reprezentuje poziom wypoczęcia postaci.



Wykres 4.9. Reprezentacja graficzna zbiorów rozmytych zawartych w zmiennej `FuzzyRest`

4.6.3.4. `FuzzyMorale`

Zmienna `FuzzyMorale` reprezentuje morale postaci. Jej wartość ostra może przyjąć wartości z przedziału $[-10, 10]$. Przy próbie zapisu wartości mniejszej bądź większej od dozwolonych zostanie przypisane odpowiednio -10 i 10.



Wykres 4.10. Reprezentacja graficzna zbiorów rozmytych zawartych w zmiennej FuzzyMorale

4.6.4. ROZMYTA MACIERZ ASOCJACYJNA

Rozmyta macierz asocjacyjna (ang. *fuzzy associative matrix*, *FAM*) jest to macierz, która ułatwia nam określenie przynależności do konkretnych zbiorów rozmytych.

```
1.  public class FAM {
2.     public FAM(FuzzyVariable row, FuzzyVariable column, string[,] actions) {
3.        list_ = new List<List<ValueTuple<float, string>>>(row.StatesCount);
4.        for ( int i = 0 ; i < row.StatesCount; ++i ) {
5.           list_.Add(new List<ValueTuple<float, string>>(column.StatesCount));
6.           for ( int j = 0 ; j < column.StatesCount ; ++j ) {
7.              column.Fuzzify(j);
8.              row.Fuzzify(i);
9.              list_[i].Add((FuzzyVariable.And(row.FuzzyValue, column.FuzzyValue),
10.                                          actions[i, j]));
11.          }
12.       }
13.    }
14.
15.    public (float val, string action) MaxValue {
16.       get {
17.          (float v, string a) value = ( float.MinValue, string.Empty );
18.          foreach ( (float val, string action) pair in list_.SelectMany(list => list) {
19.             if (pair.val > value.v)
20.                value = pair;
21.          }
22.
23.          return value;
24.       }
25.    }
26.
27.    public (float val, string action) this[int i, int j] {
28.       get { return list_[i][j]; }
29.       set { list_[i][j] = value; }
30.    }
31.
32.    private readonly List<List<(float val, string action)>> list_;
33. }
```

Powstaje ona na bazie dwóch zmiennych rozmytych. Składa się z liczby kolumn równej liczbie zbiorów zawartych w pierwszej zmiennej, a jej liczba wierszy odpowiada liczbie zbiorów zawartych w drugiej zmiennej. W poszczególnych komórkach znajduje się krotka składająca się ze stopnia przynależności w postaci zmiennej `float val` oraz nazwy zbioru rozmytego, do jakiego przynależy ta funkcja przynależności μ, w postaci zmiennej `string action`. Wartości te obliczane są w konstruktorze, który jako parametry przyjmuje dwie zmienne rozmyte oraz tablicę, w której przechowywane są odpowiadające poszczególnym komórkom nazwy zbiorów. Wewnątrz, w zagnieżdżonej pętli, obliczana jest wartość stopnia przynależności na podstawie operatora AND. Klasa udostępnia także tak zwany indekser. Język C# nie umożliwia wprost przeładowania operatora `[]`, jednak jego odpowiednikiem jest indekser, który w przeciwieństwie do znanej z języka C++ konstrukcji, może przyjąć dowolną liczbę parametrów. Zwraca on w tym wypadku odpowiednią krotkę z macierzy. We własności `MaxValue` znajdowana jest i zwracana krotka z największym stopniem przynależności do swojego zbioru.

4.6.5. Implementacja reguł rozmytych

Wybrałem implementację reguł rozmytych opartą na macierzach asocjacyjnych głównie z powodu prostoty implementacji.

W mojej grze silnik logiki rozmytej decyduje o akcjach podejmowanych zarówno przez osadników, jak i atakujących. Podejmowanie decyzji ma miejsce odpowiednio w klasie kolonii lub oblegających w funkcji `MakeDecision`, która jest przeciążona dla różnych zmiennych rozmytych. Jako argumenty przyjmuje dwie zmienne rozmyte, dla których budowana jest rozmyta macierz asocjacyjna i zwracana jest nazwa zbioru, dla którego obliczono największy stopień przynależności.

W przypadku kolonistów określanie zadania jest dwuetapowe. Najpierw na podstawie zmiennych `FuzzyRest` i `FuzzyLaziness` określane jest, czy osadnik jest na tyle zmęczony, żeby pójść spać, czy powinien robić coś innego. Tę regułę będę dalej nazywać regułą spania. Następnie na podstawie `FuzzyHP` i `FuzzyLaziness` określane jest, czy osadnik powinien pracować – pobiera wtedy z kolejki priorytetowej pracę o największym priorytecie – czy udać się na odpoczynek. Tę regułę będę dalej nazywać regułą pracy.

		FuzzyLaziness		
		Lazy	Average	HardWorking
FuzzyRest	Tired	Sleep	Sleep	DoSth
	Normal	Sleep	DoSth	DoSth
	FullOfEnergy	DoSth	DoSth	DoSth

Tabela 4.3. Wybór zachowania na podstawie `FuzzyRest` i `FuzzyLaziness`. Nazwy zgodne z wykorzystanymi w aplikacji.

		FuzzyLaziness		
		Lazy	Average	HardWorking
FuzzyHP	Dying	Rest	Rest	Rest
	Low	Rest	Rest	Work
	Average	Rest	Work	Work
	High	Work	Work	Work
	Full	Work	Work	Work

Tabela 4.4. Wybór zachowania na podstawie `FuzzyHP` i `FuzzyLaziness`. Nazwy zgodne z wykorzystanymi w aplikacji.

Natomiast w przypadku atakujących rodzaj akcji wybierany jest na podstawie jednej macierzy, która budowana jest na podstawie zmiennych: `FuzzyHP` oraz `FuzzyMorale`. Decydowane jest, czy dany oblegający oddali się na spoczynek, będzie dalej atakować, czy może postanowi uciekać. Tę regułę będę dalej nazywać regułą ucieczki-ataku.

		FuzzyMorale				
		Broken	Low	Average	High	Full
FuzzyHP	Dying	Flee	Flee	Flee	Flee	Rest
	Low	Flee	Flee	Flee	Rest	Attack
	Average	Flee	Flee	Rest	Attack	Attack
	High	Flee	Flee	Attack	Attack	Attack
	Full	Flee	Attack	Attack	Attack	Attack

Tabela 4.5. Wybór zachowania na podstawie `FuzzyHP` i `FuzzyMorale`. Nazwy zgodne z wykorzystanymi w aplikacji.

4.6.5.1. PRZYKŁAD OBLICZEŃ REGUŁ ROZMYTYCH

Najpierw przedstawię proces rozumowania rozmytego dla kolonisty o następujących wartościach zmiennych rozmytych:

- `FuzzyHP` – 47,5, dla `MaxHP` – 60;
- `FuzzyLaziness` – 6,75;
- `FuzzyRest` – 6,33.

Na początku obliczana jest reguła spania, dla której trzeba obliczyć następne stopnie przynależności do zbiorów:

- `FuzzyRest.Tired` – 0,25;
- `FuzzyRest.Normal` – 0,75;
- `FuzzyRest.FullOfEnergy` – 0;
- `FuzzyLaziness.Lazy` – 0;
- `FuzzyLaziness.Average` – 0,625;
- `FuzzyLaziness.HardWorking` – 0,375.

Następnie kolejno przeprowadzane są operacje AND, aby wyznaczyć wartości poszczególnych komórek macierzy.

- `Tired` AND `Average` – $0{,}25 \text{ AND } 0{,}625 = 0{,}25$
- `Normal` AND `Average` – $0{,}75 \text{ AND } 0{,}625 = 0{,}625$
- `Normal` AND `HardWorking` – $0{,}75 \text{ AND } 0{,}375 = 0{,}375$
- `Tired` AND `HardWorking` – $0{,}25 \text{ AND } 0{,}375 = 0{,}25$

Dla pozostałych reguł wartości wynoszą zero. Po tych obliczeniach macierz odpowiadająca tej regule wygląda następująco:

		FuzzyLaziness		
		Lazy	Average	HardWorking
FuzzyRest	Tired	0 – Sleep	0,25 – Sleep	0,25 – DoSth
	Normal	0 – Sleep	0,625 – DoSth	0,375 – DoSth
	FullOfEnergy	0 – DoSth	0 – DoSth	0 – DoSth

Tabela 4.6. Stopnie przynależności wraz z nazwami zbiorów wyliczone dla reguły spania

Wybieramy z tej macierzy krotkę, w której stopień przynależności jest największy. Wynosi on 0,625, a akcją jest *DoSth*, która odpowiada temu, że osadnik nie idzie spać i przechodzi do podejmowania następnej decyzji.

Następnie wyliczana jest reguła pracy, dla której trzeba dodatkowo obliczyć następujące stopnie przynależności do zbiorów:

- `FuzzyHP.Dying` – 0;
- `FuzzyHP.Low` – 0;
- `FuzzyHP.Average` – 0;
- `FuzzyHP.High` – 0,25;
- `FuzzyHP.Full` – 0,75.

Daje nam to następujące dalsze operacje:

- `High` AND `Average` – $0{,}25 \text{ AND } 0{,}625 = 0{,}25$
- `Full` AND `Average` – $0{,}75 \text{ AND } 0{,}625 = 0{,}625$
- `Full` AND `HardWorking` – $0{,}75 \text{ AND } 0{,}375 = 0{,}375$
- `High` AND `HardWorking` – $0{,}25 \text{ AND } 0{,}375 = 0{,}25$

Dla pozostałych reguł wartości wynoszą zero. Po tych obliczeniach macierz odpowiadająca tej regule wygląda następująco:

		FuzzyLaziness		
		Lazy	Average	HardWorking
FuzzyHP	Dying	0 – Rest	0 – Rest	0 – Rest
	Low	0 – Rest	0 – Rest	0 – Work
	Average	0 – Rest	0 – Work	0 – Work
	High	0 – Work	0,25 – Work	0,25 – Work
	Full	0 – Work	0,625 – Work	0,375 – Work

Tabela 4.7. Stopnie przynależności wraz z nazwami zbiorów wyliczone dla reguły pracy

Wybieramy z tej macierzy krotkę, w której stopień przynależności jest największy. Znów wynosi on 0,625, a akcja tym razem to *Work*. Odpowiada ona temu, że osadnikowi zostanie przypisana praca o najwyższym priorytecie z kolejki.

Następnie przedstawię proces rozumowania rozmytego dla oblegającego o następujących wartościach zmiennych rozmytych:

- `FuzzyHP` – 37,5, dla `MaxHP` – 60;
- `FuzzyModale` – 2,25.

Dla atakującego wyliczana jest reguła ucieczki-ataku, dla której trzeba obliczyć następujące stopnie przynależności do zbiorów:

- `FuzzyHP.Dying` – 0
- `FuzzyHP.Low` – 0
- `FuzzyHP.Average` – 0,25;
- `FuzzyHP.High` – 0,75;
- `FuzzyHP.Full` – 0;
- `FuzzyMorale.Broken` – 0;
- `FuzzyMorale.Low` – 0;
- `FuzzyMorale.Average` – 0,75;
- `FuzzyMorale.High` – 0,25;

- `FuzzyMorale.Full` – 0;

Daje nam to następujące operacje:

- **Average** AND **Average** – 0,25 AND 0,75 = 0,25
- **High** AND **Average** – 0,75 AND 0,75 = 0,75
- **High** AND **High** – 0,75 AND 0,25 = 0,25
- **Average** AND **High** – 0,25 AND 0,25 = 0,25

Dla pozostałych reguł wartości wynoszą zero. Po tych obliczeniach macierz odpowiadająca tej regule wygląda następująco:

		FuzzyMorale				
		Broken	Low	Average	High	Full
FuzzyHP	Dying	0 – Flee	0 – Flee	0 – Flee	0 – Flee	0 – Rest
	Low	0 – Flee	0 – Flee	0 – Flee	0 – Rest	0 – Attack
	Average	0 – Flee	0 – Flee	0,25 – Rest	0,25 – Attack	0 – Attack
	High	0 – Flee	0 – Flee	0,75 – Attack	0,25 – Attack	0 – Attack
	Full	0 – Flee	0 – Attack	0 – Attack	0 – Attack	0 – Attack

Tabela 4.8. Stopnie przynależności wraz z nazwami zbiorów wyliczone dla reguły ucieczki-ataku

Wybieramy z tej macierzy krotkę, w której stopień przynależności jest największy. Wynosi on 0,75, a akcja to *Attack*. Odpowiada ona temu, że atakujący wylosuje jednego z kolonistów, zacznie zmierzać w jego kierunku, a kiedy do niego dotrze, zacznie atakować.

5. Opis wykorzystanych narzędzi

5.1. SFML.Net

SFML (ang. *Simple and Fast Multimedia Library*) jest to, jak sama nazwa wskazuje, prosta i szybka biblioteka multimedialna. Zapewnia prosty interfejs do różnych komponentów komputera, takich jak mysz, klawiatura, ekran, dźwięk czy sieć[35]. Podstawowa wersja biblioteki jest napisana w języku C++ i obecnie dostępną wersją jest SFML $v2.4.1$[36]. Posiada też jednak oficjalne wersje dla języka C i platformy .Net, a także nieoficjalne dla większości popularnych języków takich jak na przykład: Python, Java, Ruby, czy nawet dla języka Pascal[37].

W swojej pracy wykorzystałem jednak nieoficjalną wersję dla platformy .Net stworzoną przez Laurenta Gomila wraz z Zachariahem Brownem[38]. Dokonałem takiego wyboru, ponieważ oficjalnie wydaną wersją na platformę Microsoftu jest SFML $v2.2$, a nieoficjalna jest dla nieco nowszej wersji – SFML $v2.3.1$. Ponadto wersja oficjalna jest dokładnym portem wersji oryginalnej z języka C++ i jedyną istotną zmianą jest wykorzystanie wbudowanej w język C# obsługi zdarzeń. Natomiast wersja nieoficjalna w dużo większym stopniu wykorzystuje specyficzne dla języka C# sposoby pisania kodu takie jak właściwości w miejsce funkcji typu: `getXXX()`, `setXXX()` czy `isXXX()`.

W oryginalnej wersji SFML dostarcza takie moduły jak: system, window, graphics, audio oraz network. Wersja dla platformy .Net została pozbawiona modułu odpowiedzialnego za łączność sieciową oraz pewnych składowych modułu system, takich jak na przykład wątki. Zostało to zapewne podyktowane tym, że w porównaniu do biblioteki standardowej C++[39], jej odpowiednik z .Netu dostarcza między innymi funkcje sieciowe, więc nie było potrzeby ich dublowania.

5.1.1. `SFML.System`

W przestrzeni nazw `System` zdefiniowane są tylko wektory dwu- i trójelementowe oraz struktura `Time` i klasa `Clock`, odpowiedzialne za pomiar i przechowywanie czasu. Są to pewne podstawowe typy.

5.1.2. `SFML.Window`

W przestrzeni nazw `Window` znajdują się klasy i struktury zapewniające obsługę myszy, klawiatur, joysticka, dotyku czy sensorów dostępnych w urządzeniu. Oczywiście oprócz tego znajduje się też klasa odpowiadająca obsłudze okien. Są tam również typy obsługujące w czasie rzeczywistym operacje wejścia/wyjścia.

5.1.3. `SFML.Graphic`

W przestrzeni nazw `Graphic` znajdują się klasy i struktury odpowiadające wszystkim obiektom, jakie można wyświetlać. Są to takie typy jak kolory, różnego rodzaju kształty, czcionki, shadery, tekstury i wiele innych. Tutaj także znajduje się interfejs `Drawable`, który pozwala narysować na ekranie dowolny stworzony przez nas obiekt.

[35] *SFML* [http://www.sfml-dev.org/]
[36] *Download SFML* [http://www.sfml-dev.org/download/sfml/2.4.1/]
[37] *Bindings SFML* [http://www.sfml-dev.org/download/bindings.php]
[38] *Graphnode SFML.Net* [https://www.nuget.org/packages/Graphnode.SFML.Net/]
[39] Od wersji C++11 biblioteka standardowa tego języka udostępnia klasy umożliwiające obsługę wątków, jednak sam SFML został stworzony wcześniej.

5.1.4. `SFML.Audio`

W przestrzeni nazw `Audio` znajdują się klasy i struktury odpowiedzialne za przechowywanie i odtwarzanie dźwięków czy plików dźwiękowych. W swojej pracy nie wykorzystywałem żadnego dźwięku, więc moja wiedza na temat tego modułu jest bardzo ograniczona.

6. PREZENTACJA GRY

W grze użyłem następujących, generowanych przy uruchomieniu, tekstur:

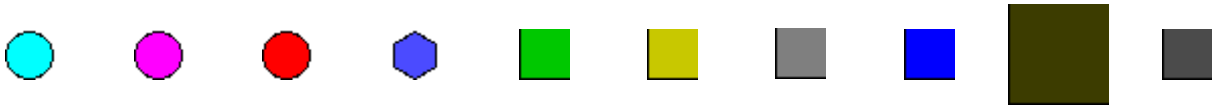
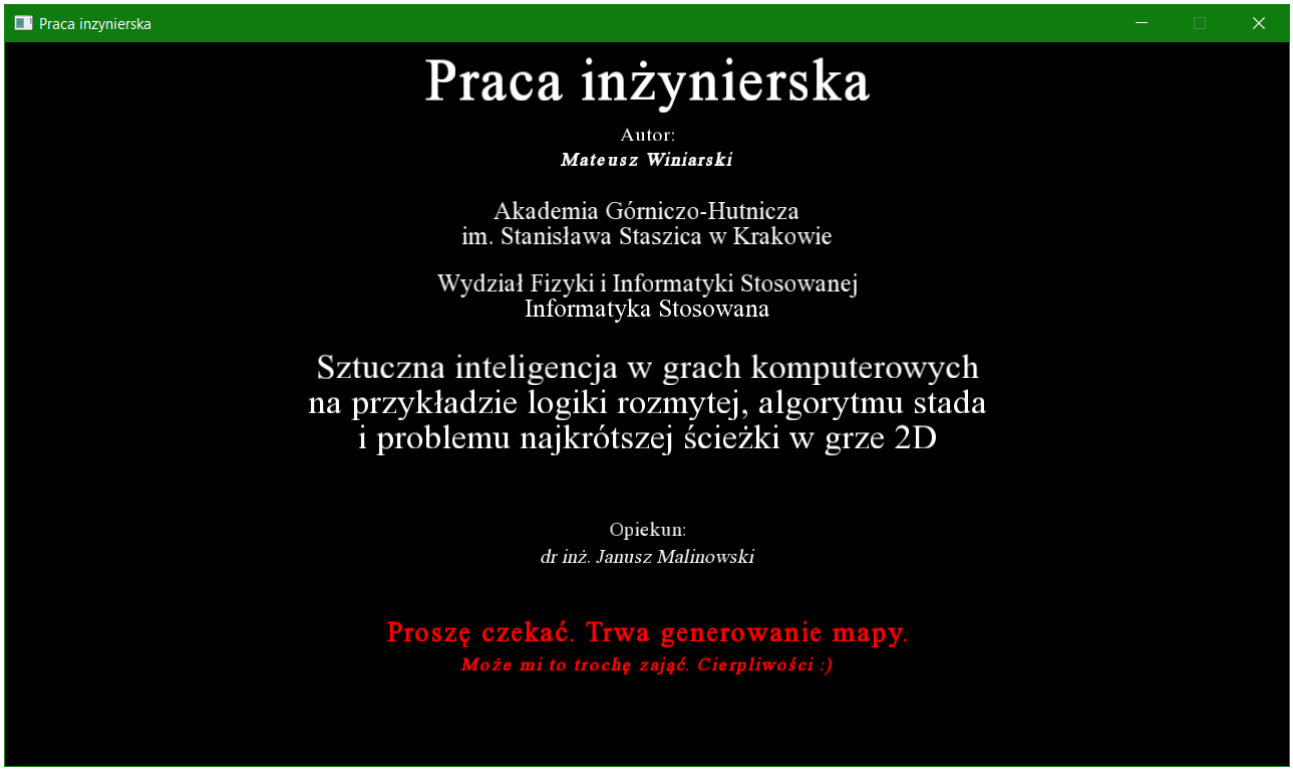


Tabela 6.1. Od lewej kolejno tekstury: kolonisty, kolonisty po zaznaczeniu przez gracza, atakującego, zwierzęcia, pola trawy, pola piasku, pola góry, łóżka, stołu oraz ściany.

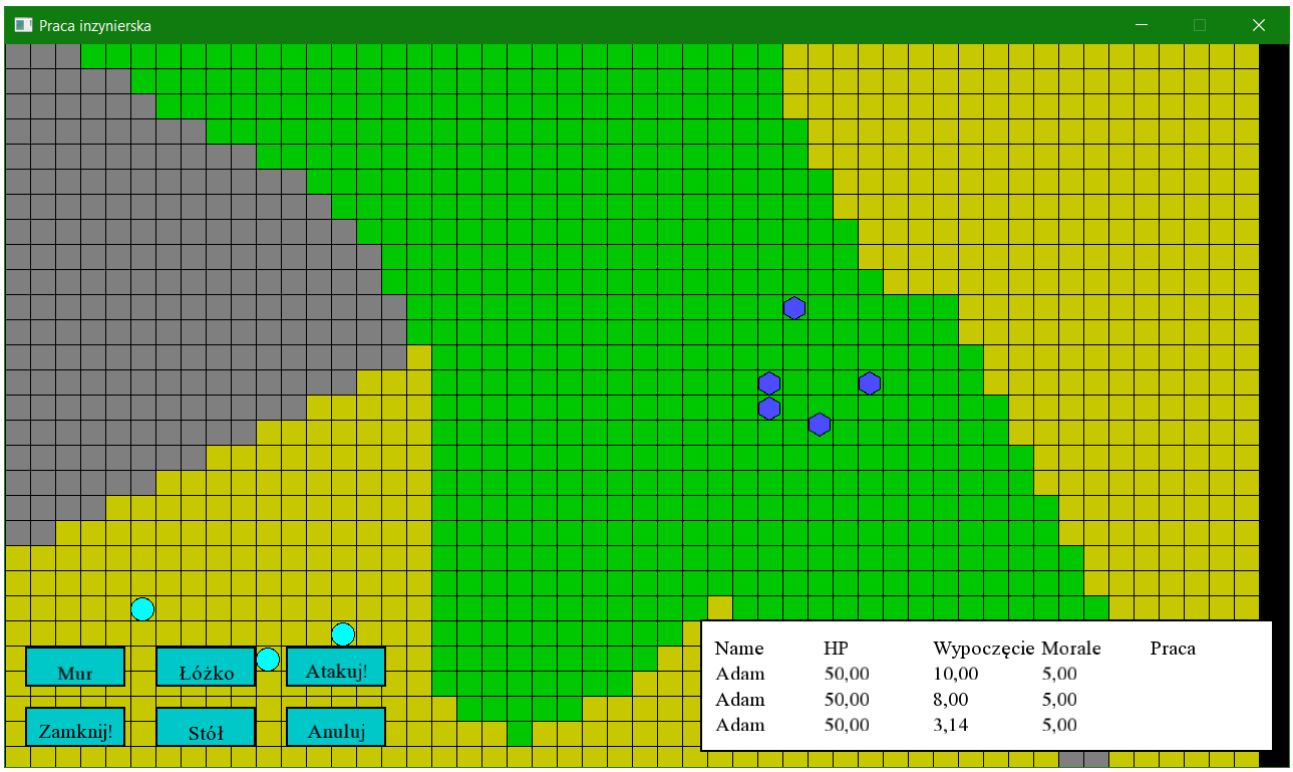
Tekstury pól mapy i konstrukcji nie są całkowicie obramowane, ponieważ uznałem, że w ten sposób plansza lepiej się prezentuje. Z kolei stół zajmuje cztery pola, więc jego tekstura jest dwukrotnie większa.

Po uruchomieniu gra otwiera się w oknie 1280×720 i wita nas ekranem startowym. W tym czasie generowane są tekstury, przygotowywana jest mapa o wielkości 50×50 pól, stado i koloniści.



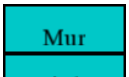
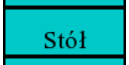
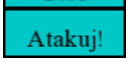
Rysunek 6.1. Ekran startowy gry

Następnie pojawia się plansza wraz ze stadem i kolonistami.

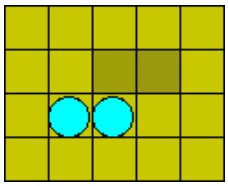


Rysunek 6.2. Widok po rozpoczęciu gry

Na rysunku powyżej widać trzech kolonistów oraz stado składające się z pięciu zwierząt. W lewym dolnym rogu znajdują się przyciski, przy pomocy których gracz może wykonywać interakcje z grą:

-  – po naciśnięciu gracz ma możliwość postawienia muru.
-  – po naciśnięciu gracz ma możliwość wybudowania łóżka.
-  – po naciśnięciu gracz ma możliwość wybudowania stołu.
-  – po naciśnięciu gracz ma możliwość wskazania zaznaczonym kolonistom ataku konkretnego oblegającego.
-  – po naciśnięciu anulowana zostanie możliwość budowy wybranego obiektu, czy ataku przeciwnika.
-  – po naciśnięciu gra zostanie wyłączona.

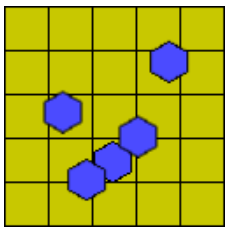
Natomiast w prawym dolnym rogu wyświetlana jest informacja o stanie kolonistów: o ich zdrowiu, wypoczęciu, morale oraz o tym, jaką pracę aktualnie wykonują.



Rysunek 6.3. Dwaj koloniści budują fragment muru. W trakcie pracy zmniejsza się ich współczynnik wypoczęcia. Wraz z postępem budowy tekstury będą zmieniać się na coraz mniej przezroczyste

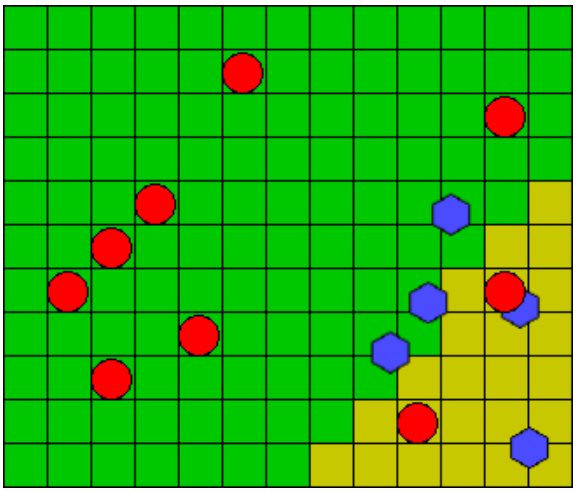
Name	HP	Wypoczęcie	Morale	Praca
Adam	50,00	8,17	5,00	Buduje
Adam	50,00	6,77	5,00	Buduje
Adam	50,00	4,76	5,00	

Rysunek 6.4. Tabela z informacjami o stanie kolonistów. Dwóch z nich pracuje przez co spada ich współczynnik wypoczęcia, trzeci natomiast odpoczywa przez co ten współczynnik mu wzrasta

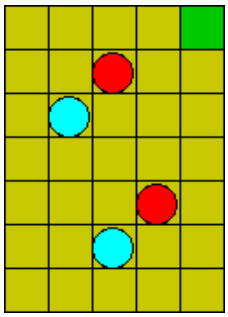


Rysunek 6.5. Przemieszczające się stado

Z przedziału $[1', 14'59"]$ losowany jest czas, za jaki ma nastąpić atak. Po jego upływie na planszy, w odległości nie mniejszej niż $^3/_5$ jej długości bądź szerokości, pojawia się od jednego do piętnastu atakujących. Znów losowany jest czas, tym razem z przedziału $[1', 4'59"]$, który atakujący wykorzystują na przygotowania, a po upływie przystępują do ataku. Jeśli wszyscy atakujący uciekną z planszy, wylosowany zostanie nowy czas, po upłynięciu którego nastąpi kolejny atak.



Rysunek 6.6. Grupa atakujących przygotowująca się do ataku



Rysunek 6.7. Koloniści walczący z atakującymi

Gra trwa, dopóki na planszy znajduje się przynajmniej jeden kolonista.

7. PODSUMOWANIE

Celem pracy było stworzenie gry komputerowej przy użyciu biblioteki SFML oraz implementacja takich algorytmów sztucznej inteligencji jak algorytm A*, algorytm stada czy logika rozmyta. W mojej ocenie założenia te zostały osiągnięte, a grę można uznać za bardzo wczesny prototyp gry polegającej na budowie osady. Widzę też dość oczywiste ścieżki jej rozwoju, takie jak dodatkowe budowle, możliwość polowań na zwierzęta czy uprawiania roślin, większa różnorodność postaci i ich cech, czy polepszenie wizualne. Można też dalej rozwijać sztuczną inteligencję.

Podczas tworzenia pracy mogłem przekonać się, w jaki sposób, od pierwszych szkiców i pomysłów aż do pierwszych działających i funkcjonalnych wersji, powstają obecnie złożone gry komputerowe. Dzięki temu jeszcze dobitniej zrozumiałem, że proces ten jest trudny, czasochłonny i bardzo złożony. Mogę także z całą pewnością stwierdzić, że najwięcej czasu zajmuje testowanie napisanego kodu i doprowadzenie do tego, żeby działał on zgodnie z oczekiwaniami. Wielu początkujących programistów zachłyśniętych pierwszymi sukcesami bagatelizuje ten etap tworzenia oprogramowania. Jednak obecnie konsument – w tym wypadku gracz – jest coraz bardziej przeczulony na punkcie wszelkich niedociągnięć czy błędów w grach. Wielu młodych programistów myśli także, że skoro napisali kod tak, jak chcieli i wydaje się im, że będzie działał tak, jak zaplanowali, to już samo to implikuje, że tak właśnie będzie. Niestety w przeważającej liczbie przypadków tak się nie dzieje, zwykle z powodu nieuwzględnienia pewnych warunków, o których wydawało się, że zajść nigdy nie mogą, czy z powodu błędnych założeń poczynionych przez programistę.

Kilkukrotnie w trakcie tworzenia tej pracy zastanawiałem się, czy nie byłoby lepiej zamiast prostej biblioteki SFML wykorzystać jakiegoś bogatego narzędzia programistycznego pokroju Unity. Jednak teraz sądzę, że w przypadku tak niewielkiego i prostego projektu, jaki stworzyłem, byłoby to porywanie się z motyką na słońce. Dzięki temu czas, który musiałbym poświęcić na poznawanie działania takiego narzędzia, mogłem w znacznej mierze przeznaczyć na pisanie aplikacji, a zapoznanie się z biblioteką sprowadziło się do przeczytania dokumentacji i własnych testów jak została ona przeniesiona na platformę .Net.

Pomimo różnych trudności i zawiłości napotkanych podczas pisania tej pracy oraz świadomości skomplikowania procesu tworzenia gier było to dla mnie zajęcie bardzo ciekawe i dające duże możliwości rozwoju oraz satysfakcję z samodzielnego tworzenia gry. W przyszłości z wielką chęcią chciałbym dopracować grę tworzoną w ramach tej pracy, jak i stworzyć inne, na które zrodziło się w tym czasie już kilka pomysłów.

Na koniec chciałbym podziękować Panu dr inż. Januszowi Malinowskiemu za udzielanie cennych wskazówek, a zwłaszcza za umożliwienie napisania pracy na ten interesujący mnie temat.

BIBLIOGRAFIA

1. Millington Ian, *Artificial intelligence for games*, San Francisco USA, Elsevier Inc., 2006, ISBN 10: 0-12-497782-0
2. Buckland Mat, *Programming Game AI by Example*, Plano USA, Wordware Publishing Inc., 2005, ISBN 1-55622-078-2
3. DeLoura Mark A., *Game Programming Gems*, t. 1, Hingham USA, Charles River Media, 2000, ISBN 1-58-450049-2
4. Bourg David M., Seeman Glenn, *AI for Game Developers*, Sebastopol USA, O'Reilly Media Inc., 2004, ISBN 0-596-00555-5
5. Reynolds Craig, *Boids* [online], aktualizacja 30.06.2007r., [dostęp: 29.12.2016r.] Dostępny w Internecie: <http://www.red3d.com/cwr/boids/>
6. Wardziński Krzysztof, *Przegląd algorytmów sztucznej inteligencji stosowanych w grach komputerowych* [online], „Homo communicativus" 2008, nr 3(5), s. 249-255, [dostęp: 29.12.2016r.] Dostępny w Internecie: <http://www.hc.amu.edu.pl/numery/5/HC5.pdf>
7. Kalwoda Jan, *Prawie wszystko o Logice Rozmytej* [online], Warszawa, Politechnika Warszawska, [dostęp: 29.12.2016r.], Dostępny w Internecie: <http://www.isep.pw.edu.pl/ZakladNapedu/dyplomy/fuzzy/index.htm>
8. Gomila Laurent, Brown Zachariah, *Graphnode.SFML.Net* [online], aktualizacja 22.08.2016r, [dostęp: 29.12.2016r.], Dostępny w Internecie: <https://www.nuget.org/packages/Graphnode.SFML.Net/>
9. Albahari Joseph, Albahari Ben, *C# 6.0 w pigułce*, Wyd. 4, przeł. Górczyński Robert, Hubisz Jakub, Piwko Łukasz, Gliwice, Helion, 2016, ISBN 978-83-283-2423-7
10. Skeet Jon, *C# od podszewki*, Wyd. 2, przeł. Grabis Janusz, Gliwice, Helion, 2012, ISBN 978-83-246-6458-0
11. Albahari Joseph, Albahari Ben, *C# 5.0 Leksykon kieszonkowy*, Wyd. 3, przeł. Szeremiota Przemysław, Gliwice, Helion, 2013, ISBN 978-83-246-6276-0
12. *C# Guide* [online], Microsoft, aktualizacja 03.08.2016r., [dostęp: 29.12.2016r.], Dostępny w Internecie: <https://docs.microsoft.com/pl-pl/dotnet/articles/csharp/>
13. *Tutorials for SFML 2.3* [online], [dostęp: 29.12.2016r.], Dostępny w Internecie: <http://www.sfml-dev.org/tutorials/2.3/>
14. *Documentation of SFML 2.3.1* [online], [dostęp: 29.12.2016r.], Dostępny w Internecie: <http://www.sfml-dev.org/documentation/2.3.1/>
15. *Heuristic* [online], Wikipedia, aktualizacja 17.12.2016r., [dostęp: 29.12.2016r.], Dostępny w Internecie: <https://en.wikipedia.org/wiki/Heuristic>
16. *Heuristic (computer science)* [online], Wikipedia, aktualizacja 04.12.2016r., [dostęp: 29.12.2016r.] Dostępny w Internecie: <https://en.wikipedia.org/wiki/Heuristic_(computer_science)>
17. *Fuzzy set* [online], Wikipedia, aktualizacja 12.12.2016r., [dostęp: 29.12.2016r.], Dostępny w Internecie: <https://en.wikipedia.org/wiki/Fuzzy_set>
18. *Historia gier komputerowych* [online], Wikipedia, aktualizacja 18.11.2016r., [dostęp: 29.12.2016r.], Dostępny w Internecie: <https://pl.wikipedia.org/wiki/Historia_gier_komputerowych>
19. *Pong* [online], Wikipedia, aktualizacja 04.05.2016r., [dostęp: 29.12.2016r.], Dostępny w Internecie: <https://pl.wikipedia.org/wiki/Pong>
20. *Pac-Man* [online], Wikipedia, aktualizacja 16.10.2016r., [dostęp: 29.12.2016r.], Dostępny w Internecie: <https://pl.wikipedia.org/wiki/Pac-Man>
21. *Εὑρίσκω* [online], Wikisłownik, aktualizacja 03.04.2016r., [dostęp: 29.12.2016r.], Dostępny w Internecie: <https://pl.wiktionary.org/wiki/εὑρίσκω>

Wszystkie grafiki użyte w aplikacji są generowane podczas uruchamiania bądź działania gry przez klasy: `MapTextures` oraz `GUITextures`.