Implementacja wybranych gier z zawodów programistycznych PIZZA na platformę CodinGame

(Implementation of chosen games from PIZZA contest to CodinGame platform)

Rafał Tatarczuk

Praca inżynierska

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Wrocław 2022

Streszczenie

Praca zawiera implementację silników dwóch wybranych gier (Owce i Malowanie) z zawodów programistycznych PIZZA na platformę CodinGame wraz z przykładowymi botami. Treści i działanie zostały dostosowane pod docelową platformę. W pracy znajduje się opis techniczny, jak i omówienie gier.

This paper contains implementation of two chosen games engines (Sheep and Painting) for CodinGame platform, chosen from programming competitions PIZZA. Implementation of example bots is also included. Both statements and game logic was adjusted for target platform. Paper contains technical description and overview of the rules.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Zawody PIZZA

Zawody PIZZA (Programistyczno-Informatyczne Zespołowe Zawody Algorytmiczne)[1] były organizowane w latach 2012-2018 w Instytucie Informatyki Uniwersytetu Wrocławskiego. Finał zawodów trwał dziesięć godzin, w ciągu których trzyosobowe drużyny pisały boty do trzech różnych gier. Zawodnicy łączyli się z serwerem gry przez TCP/IP i tam wysyłali komendy zgodne ze specyfikacją. Każda gra była podzielona na turnieje i rundy, pomiędzy którymi zawodnicy mogli usprawniać swoje algorytmy. W związku z tym, że boty nie były uruchamiane na serwerze – tylko na komputerach zawodników, mieli oni pełną dowolność w wyborze języka programowania, środowiska oraz technologii, w których tworzyli boty. Dzięki wykorzystaniu platformy Kubernetes w trakcie zawodów była możliwość rozgrywania bardzo dużej ilości gier jednocześnie. Większość edycji zawodów PIZZA odbywała się w języku polskim. Tylko ostatnia edycja miała rangę międzynarodową.

1.2. Cel Pracy

Celem tej pracy jest implementacja gier z zawodów PIZZA na międzynarodową platformę CodinGame [2], na której znajdują się różne gry, w których każdy może spróbować swoich sił. Wiele gier z finałów zawodów PIZZA było interesujących pod względem strategicznym, a niektóre z nich można przystosować pod platformę CodinGame . W niniejszej pracy zostały zaimplementowane silniki dwóch wybranych gier. Zarówno polecenia, zasady jak i przebieg rozgrywki zostały dostosowane do wyżej wymienionej platformy. Do implementacji silników zostały również dołączone przykładowe implementacje botów.

Rozdział 2.

Wybrane gry i ich dostosowanie do nowej platformy

2.1. Różnice pomiędzy środowiskiem zawodów PIZZA a platformą CodinGame

Dzięki wykorzystaniu narzędzia Kubernetes[3] w trakcie finału możliwe było rozgrywanie dużej ilości gier na raz. Zawodnicy musieli uruchamiać wiele instancji swoich programów lub obsługiwać wiele połączeń, by wydawać komendy dla wielu rozgrywek jednocześnie. Każdy turniej był rozgrywany niezależnie. Parametry gier były takie same dla wszystkich gier w obrębie jednego turnieju. O zwycięstwie decydowała ważona suma punktów ze wszystkich turniejów. W czasie rozgrywki większość zespołów tworzyła własne wizualizacje gier, aby łatwiej dostrzegać miejsca w algorytmie wymagające usprawnienia.

Na platformie CodinGame każda gra odbywa się niezależnie. Gracze wysyłają kody źródłowe, które wczytują ze standardowego wejścia dane zgodne ze specyfikacją, a następnie po wykonaniu obliczeń wypisują komendy, również zgodne ze specyfikacją na standardowe wyjście. Następnie kod uruchamiany jest na platformie (CodinGame posiada wsparcie dla wielu języków programowania). Po zakończeniu rozgrywki generowana i wyświetlana na stronie jest wizualizacja pojedynku. Użytkownicy platformy nie mają limitów ilościowych i czasowych w nadsyłaniu programów, mogą udoskonalać swoje rozwiązania przez cały czas. Boty użytkowników konkurują między sobą. Ranking jest cały czas uaktualniany i tworzony na podstawie ciągłych zestawień botów. Na CodinGame również bardzo ważną kwestią jest wizualizacja, która jest nieodłączną częścią gier na tej platformie. Ułatwia ona graczom zrozumienie zasad i ilustruje działanie ich algorytmów. Wiele gier udostępnia dodatkowe informacje w trakcie wizualizacji. Wymóg przedstawienia gier w formie graficznej niesie za sobą różne ograniecczenia.

2.1.1. Typy gier

Na zawodach PIZZA można było wyróżnić dwa znacząco różne typy gier. W przypadku obu typów jedna rozgrywka była podzielona na wiele tur.

Jatka (ang. free-for-all)

Pierwszy typ stanowią gry, w których zestawiane były wszystkie boty uczestników jednocześnie (w jednej rozgrywce). W finałach brało udział około trzydzieścipięć zespołów, więc zestawienie ze sobą takiej ilości algorytmów nie sprawiało żadnych problemów. Przykładem takiej gry są na przykład „Karawany" z roku 2016.

W realiach CodinGame implementacja takich gier nie jest niestety możliwa. W niektórych grach dostępnych na platformie swoich sił próbuje nawet sto pięćdzieśiąt tysięcy graczy. Uruchomianie tak wielu botów wymagałoby ogromnych zasobów, biblioteka z bazowym silnikiem na platformie nie udostępnia takiej możliwości. Przy wyborze, wszystkie gry tego typu musiały zostać odrzucone z wyżej wymienionego powodu.

Jeden na jednego

Drugi typ to konkurencje pomiędzy dwoma zespołami. W trakcie pojedyńczego turnieju każde dwa zespoły były ze sobą zestawiane i na podstawie wyników z tych rozgrywek przyznawane były ponktu za dany turniej.

To właśnie do takiego rodzaju rozgrywek przystosowany jest silnik CodinGame. Umożliwia on zestewianie od dwóch do ośmiu algorytmów, uruchamianych podczas jednej partii. Wszystkie gry, w których zgodnie z treścią mierzyły się dwa boty, były więc dobrymi kandydatami do przystosowania pod silnik CodinGame.

2.1.2. Pojedyncza rozgrywka

Na zawodach PIZZA komunikacja między urządzeniem zawodnika (na którym uruchomiony był algorytm), a serwerem gry odbywała sie przez TCP/IP. Opis aktualnego stanu świata gry był wysyłany wyłącznie na żądanie. Wysyłanie informacji takich jak opis całej planszy wraz z jej aktualnym stanem wymagał większego nakładu mocy obliczeniowej, dlatego uczestnicy mogli wysyłać żądania tego typu raz na kilkadziesiąt tur. Do pobierania innych, mniejszych informacji – jak na przykład opis jednostek, udostępnione były inne żądania. Do sterowania swoimi pionkami gracze wysyłali na serwer różne komendy zgodne ze specyfikacją. Warto zaznaczyć, że z perspektywy serwera polecenia od uczestników zawodów nie były niezbędne do prawidłowego przebiegu gry. Gdy w trakcie tury nie nadeszła żadna komenda, jednostki danego gracza były bezczynne. Wspólna dla wszystkich gier była komenda **WAIT**,

służąca do synchronizacji pomiędzy turami. Żądanie od razu zwracało **OK**, a w momencie rozpoczęcia przez serwer nowej tury zwracało: **OK NEXT_TURN** lub po prostu **OK**. Dodatkowo, w przypadku niektórych gier w kolejnych liniach pojawiały się informacje na temat poprzedniej i nowo rozpoczętej tury uznane przez twórców za niezbędne . Pojedynczy pojedynek między botami miał czasem tysiące tur i mógł trwać nawet ponad kilka minut.

Na platformie CodinGame pojedyncza gra wygląda zupełnie inaczej. W specyfikacji każdej gry jast dokładny opis danych wejściowych, które muszą zostać wczytane i oczekiwanego wyjścia wypisanego przez algorytm. Napisany program na początku wczytuje dane inicjalizacyjne (zazwyczaj niezmienniki) i pełen opis planszy. Następnie w nieskończonej pętli wczytuje informacje na temat aktualnej tury, a po wykonaniu obliczeń wypisuje wskazaną w poleceniu liczbę komend dla swoich jednostek. Niewczytanie wszystkich danych spowoduje desynchronizację wejścia, a nie wypisanie żądanej liczby komend kończy się przekroczeniem czasu (**timeout!**), co skutkuje przegraną. Jedna rozgrywka na CodinGame może trwać maksymalnie trzydzieści sekund i zazwyczaj zamyka się w kilkuset turach.

Sposób w jaki działają gry różni się znacząco pomiędzy tymi dwoma środowiskami, w związku z czym treści z zawodów PIZZA musiały być dostosowane pod CodinGame.

2.2. Malowanie

W grze „Malowanie”[4] z roku 2016 gracze mają kolorowe pionki, które potrafią kolorować elementy gry tj. np. pola planszy lub inne pionki. Celem gry jest pokrycie swoim kolorem farby większej powierzchni niż przeciwnik lub przejęcie wszystkich pionków przeciwnika przez pokrycie swoim kolorem większej ilości pól które wchodzą w ich skład. Ta gra wydawała się bardzo dobrym kandydatem na implementację przy użyciu silnika CodinGame ze względu na ciekawe i unikalne zasady. Nie udało mi się również znaleźć na platformie zbliżonej pod kątem reguł i już zaimplementowanej gry.

Dostosowanie zasad polegało głównie na zmniejszeniu ogólnej wielkości plansz oraz ustaleniu stałej wielkości pionków graczy (kwadrat o boku długości trzy). Te zmiany przystosowały „Malowanie” do limitów długości gier. Dodatkowo dodany został prosty generator mapy. Na finałach oryginalnych zawodów jedna mapa przypadała na jeden turniej. W przypadku CodinGame użytkownicy mogliby wykorzystać stałą pulę map, opierając na niej działanie swojego algorytmu. Dane wejściowe przedstawione graczom w pętli gry również zostały zmodyfikowane.

2.3. Owce

„Owce”[5] to gra z roku 2017, w której gracze sterowali pasterzami i psami pasterskimi. Celem było zdobycie większej ilości wełny niż przeciwnik. Odbywało się to przez strzyżenie owiec i składowanie wełny w swoich szopach - przy użyciu „pasterzy”. Psy pasterskie pomagały manewrować stadami owiec przemieszczającymi się po planszy. Sposób sterowania jednostkami jest dosyć standardowy, ale ze względu na ciekawą i bardzo nietypową mechanikę uznałem „Owce” za dobry wybór do implementacji.

Zasady gry nie wymagały dużych zmian przed implementacją na CodinGame. Tak jak w przypadku „Malowania” rozmiar planszy oraz ilość ruchomych jednostek zostały zmniejszone. W pętli gry każdy gracz otrzymuje informację na temat wszystkich jednostek (psów, pasterzy i szop). Dodatkowo został zmieniony sposób indeksowania pionków na typowy dla docelowej platformy, aby ułatwić implementację jej użytkownikom.

Rozdział 3.

Środowisko

Implementacje na platformę CodinGame składają się z dwóch głównych części:

- silnik i tak zwany sędzia, zaimplementowany z użyciem języka **Java**
- wizualizacja dla użytkowników napisana w większości z użyciem **JavaScript**'u

Dla ułatwienia pracy programisty, narzędzia udostępniany przez CodinGame pozwalają niemal całkowicie pominąć pisanie kodu wizualizacji. Dodatkowo do treści zadań wykorzystywany jest **HTML**, a do generowania szablonów rozwiązań w różnych językach "stub generator". W momencie pisania tej pracy CodinGame wspiera aż 27 języków w różnych wersjach. Tabelę ze wspieranymi językami można znaleźć tutaj: [6].

3.1. Java

Baza silnika gier i sędzia są zaimplementowane przy użyciu Javy w wersji JDK 8[7] z roku 2014. Jest to już dość nieaktualna wersja biorąc pod uwagę najnowsze wydanie LTS: JDK 17 z września roku 2021, co implikuje brak wielu przydatnych narzędzi, metod i usprawnień.

Twórcy udostępniają bazę gry pod nazwą CodinGame SDK poprzez publiczne repozytorium na platformie GitHub[8]. Dostępne są tam również przykładowe implementacje gier, które znacznie ułatwiają zrozumienie sposobu działania silnika. Udostępniona przez twórców dokumentacja pokrywa jedynie najważniejsze aspekty.

Silnik podzielony jest na moduły, które wystarczy zaimportować w razie chęci z wykorzystania ich. Niniejsza praca korzysta z pięciu następujących modułów:

core[9]

Jest to bazowy moduł zawierający „serce" gry – czyli klasę **GameManager**, która odpowiada za przebieg całej rozgrywki, posiada aktualny stan gry i odpowiada za komunikację z kodem napisanym przez użytkowników.

module-entities[10]

Moduł ten odpowiada za tworzenie wizualizacji. Udostępnia wiele gotowych kształtów i przydatnych narzędzi takich jak wyświetlanie tekstu. Umożliwia także używanie własnych grafik.

runner[11]

Moduł „runner" nie odpowiada za nic w wersji gry umieszczonej na platformie CodinGame . Jest to raczej narzędzie ułatwiające rozwijanie narzędzia. Pozwala na uruchomienie gry w środowisku programisty z pominięciem docelowej platformy. Posiada także podgląd danych diagnostycznych.

module-tooltip[12]

Ta część odpowiada ze dodanie „podpowiedzi" do wizualizacji. Gdy gracz najedzie kursorem na obiekt posiadający „tooltip" wyświetlone zostaną dodatkowe informacje o tym obiekcie.

module-endscreen[13]

Najprostszy z modułów. Umożliwia on dodanie ekranu końcowego z wynikami na koniec wizualizacji.

3.1.1. Maven

Do znajdowania i pobierania niezbędnych modułów i zależności wykorzystany jest Apache Maven[14], który na podstawie obiektów modelu projektu (**POM**) jest w stanie pobrać niezbędne biblioteki oraz skompilować i zbudować projekt w odpowiedniej wersji.

3.1.2. Google Guice

Jedną z najbardziej interesujących bibliotek używanych przez CodinGame SDK jest **Google Guice** [15]. Jest to bardzo lekka biblioteka do wstrzykiwania zależności w kodzie. Umożliwia stworzenie drzewiastej hierarchii iniektorów, których dokładne działanie dodatkowo jest opisywane przez dewelopera w odpowiednich klasach. Używając Guice'a możemy dodać adnotację do konstruktora lub bezpośrednio

do pola. Dzięki temu – w przeciwieństwie do najpopularniejszego javowego framework'a: Spring – Google Guice rozwiązuje cykliczne zależności bez większego nakładu pracy programisty.

3.2. JavaScript

Do wizualizacji gier w przeglądarce silnik wykorzystuje język JavaScript. Silnik napisany w Javie generuje sekwencję zmian stanów poszczególnych obiektów świata gry i na podstawie tych właśnie danych wyświetlana jest wizualizacja w przeglądarce. Najmocniej wykorzystywana jest biblioteka PixiJS[16] (do wyświetlania przeróżnej dwuwymiarowej zawartości). Twórcy biblioteki twierdzą, że dzięki natywnemu użyciu WebGL jako renderer, PixiJS jest najszybszą dostępną biblioteką.

3.3. Stub generator

Ciekawym narzędziem wartym wspomnienia jest generator szablonów rozwiązań [17], który na podstawie pseudokodu jest w stanie wygenerować taki szablon dla każdego języka programowania wspieranego przez platformę CodinGame . Stub wspiera wczytywanie i wypisywanie (`read` i `write`), podstawowe typy (jak `int`, `float`, `long` i typy tekstowe `word(<length>)`). Jest również w stanie wygenerować proste pętle przy użyciu `loop` lub `gameloop`.

Rozdział 4.

Szczegółowe zasady gier

W tym rozdziale chciałbym opisać szczegółowe reguły zaimplementowanych gier po dostosowaniu pod platformę CodinGame.

4.1. Malowanie

4.1.1. Opis rozgrywki

Celem rozgrywki jest zamalowanie większej ilości pól na planszy niż przeciwnik bądź przejęcie wszystkich pionków przeciwnika. Pojedyncza gra trwa maksymalnie 250 tur.

Gracze rywalizują w parach. Gra odbywa się na prostokątnej planszy, której rozmiary podane są na wejściu. Plansza jest podzielona na pola 1x1. Niektóre z tych pól zajmują ściany. Na początku rozgrywki każdy z graczy posiada taką samą liczbę pionków na planszy. Każdy pionek jest reprezentowany przez kwadrat o boku długości trzy. Pionki nie mogą nachodzić na ściany ani na inne pionki, muszą również w całości znajdować się w obrębie planszy.

Każdy gracz posiada swój kolor. Na starcie wszystkie pola na planszy są niepokolorowane. Nie można malować ścian.

Każdy z pionków składa się z dziewięciu kolorowych pól. Na początku rozgrywki wszystkie pola pionka są w kolorze właściciela. Pola na pionku również mogą zostać pokolorowane, w efekcie czego pionek może zmieniać właściciela. Kontrolę nad pionkem posiada gracz, którego kolor jest na większej powierzchni pionka.

Pionki mogą się poruszać i strzelać pasami farby. Właściciel zna dokładnie stan swoich pionków (położenie, kolory), nie ma jednak informacji o pionkach przeciwnika. Obaj gracze dostają pełne informacje o kolorach planszy. Jeśli nad danym polem planszy stoi pole pionka dostają informację o kolorze pola pionka.

Każdy pionek ma zbiornik z farbą z pewną ilością początkową. Może strzelać pasem farby w czterech kierunkach (góra, dół, prawo, lewo), zaczynając od środkowego pola z odpowiedniej strony (bez tego pola). Wystrzelenie pasa farby o długości **x** zużywa **x** jednostek farby ze zbiornika. Pod koniec rundy do zbiornika dodawana jest ilość farby odpowiadająca liczbie pól na planszy w kolorze pionka, nad którymi dany pionek się znajduje.

4.1.2. Opis ruchów

W trakcie jednej tury gracze wydają pionkom polecenia poruszania lub wystrzału pasa farby. Na koniec tury po zebraniu wszystkich komend silnik gry wykonuje je.

Najpierw wykonują się akcje ruchu. Wszystkie poruszające się pionki przesuwane są jednocześnie, w wyniku czego mogą nastąpić kolizje. Aby je rozwiązać, silnik gry zbiera informację o wszystkich pionkach, które kolidują ze ścianami, brzegami planszy bądź innymi pionkami. Ruchy wszystkich kolidujących pionków są cofane do stanu z poprzedniej rundy. Istnieje możliwość, że po tej operacji dalej występują jakieś kolizje, dlatego silnik powtarza operację rozwiązywania kolizji, dopóki jakiekolwiek są obecne.

Następnie wykonywane są komendy strzelania farbą. Jest możliwe że pas farby będzie krótszy niż w podanej komendzie jeśli:

- pas farby zostanie zatrzymany przez ścianę lub brzeg planszy. Nie zmienia to ilości pobranej farby ze zbiornika.
- pas farby nie może być dłuższy niż podana maksymalna długość pasa farby.
- długość pasa farby podana w komendzie była większa niż ilość jednostek farby w zbiorniku. W tym przypadku długość pasa będzie równa liczbie jednostek farby w zbiorniku.

Dodatkowo:

- jeśli w jednej turze jakieś pole zostanie pokryte dwoma kolorami, nie zmienia ono koloru.
- jeśli na polu, które znajduje się pod pasem wystrzelonej farby znajduje się pionek, pole na pionku zostaje pomalowane, a pole na planszy nie zmienia koloru.

4.1.3. Wejście

Dane inicjalizacyjne

Przed pierwszą turą każdy z graczy otrzymuje następujące informacje:

- color – numer identyfikacyjny koloru danego gracza
- rows – liczbę wierszy planszy
- cols – liczbę kolumn planszy
- maxShootDist – maksymalną długość wystrzelonego pasa farby.
- pełen opis planszy w formie napisów o długości równej liczbie kolumn i ilości równej liczbie wierszy. Każdy znak napisu oznacza jedną komórkę planszy.

.

Dane co turę

Na początku każdej tury gracz dostaje poniższe informacje:

- diffs – liczbę komórek które zmieniły swój kolor od ostatniej tury
- opis tych komórek w formie współrzędnych i nowego koloru
- pawns – liczbę kontrolowanych pionków
- opis każdego pionka za pomocą czterech liczb, z czego pierwsza to jego identyfikator, dwie kolejne to współrzędne, a czwarta to liczba jednostek farby w zbiorniku
- opis pokolorowania pionków w formie trzech napisów o długości trzech znaków dla każdego pionka, gdzie jeden znak odpowiada jednemu polu na pionku

.

4.1.4. Komendy

Dla każdego z kontrolowanych pionków gracz musi wydać jedną z dwóch komend:

MOVE

Jest to polecenie ruchu. Przyjmuje ono dwie liczby:

- id – numer identyfikacyjny pionka
- direction – cyfrę z przedziału [1–4] odpowiadającą kierunkowi. Kolejno: góra, dół, prawo, lewo

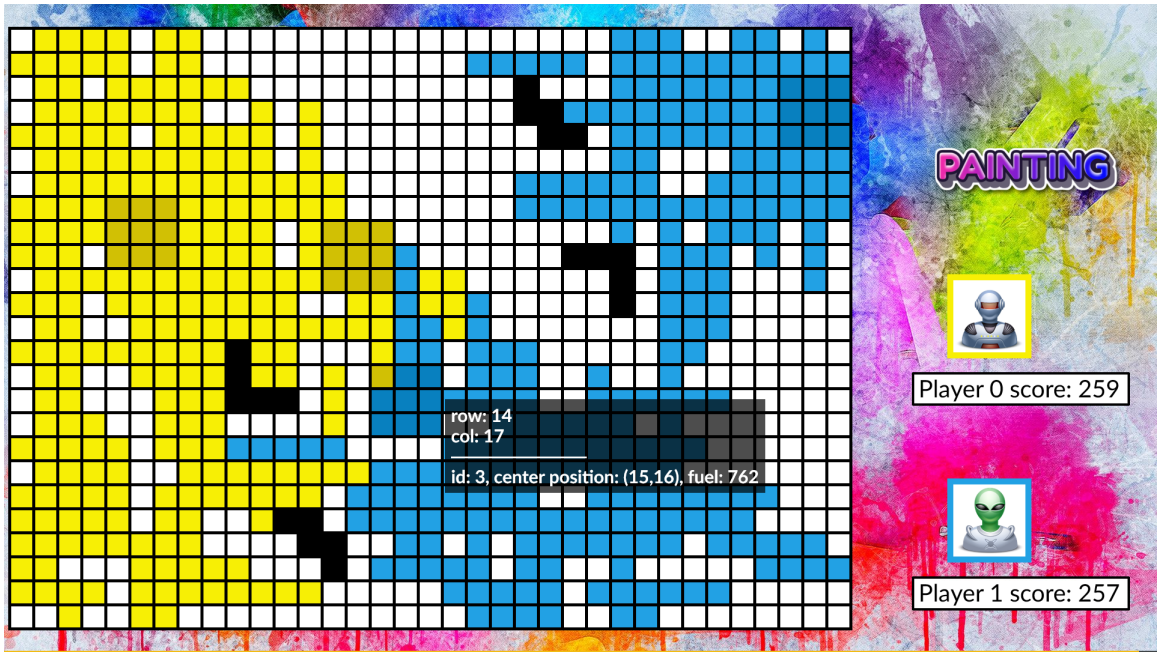
SHOOT

Jest to polecenie wystrzelenia pasa farby. Przyjmuje ono trzy liczby:

- id – numer identyfikacyjny pionka
- direction – cyfrę z przedziału [1–4] odpowiadającą kierunkowi. Kolejno: góra, dół, prawo, lewo
- length – długość pasa farby

4.1.5. Opis wizualizacji

Na poniższym obrazku znajduje się przykładowa klatka z wizualizacji. Kolorem białym oznaczone są jeszcze niepokolorowane pola. Czarny kolor reprezentuje ściany. Kolory żółty i niebieski są kolorami graczy. Pomalowane przez nich pola panszy są jaśniejsze niż kolory na pionkach. Można również zaobserwować, że jeden z niebieskich pionków został „postrzelony" farbą żółtą. Obok niego znajduje się pasek podpowiedzi, uzyskany za pomocą modułu **tooltip**. Po prawej stornie znajdują się aktualne wyniki obu graczy.



Rysunek 4.1: Przykładowa klatka z wizualizacji „Malowania"

4.2. Owce

4.2.1. Opis rozgrywki

Celem rozgrywki jest zgromadzenie na koniec dwustu pięćdziesięciu–turowej gry większej ilości wełny niż przeciwnik.

W pojedynczej grze bierze udział dwóch graczy. Rywalizują ze sobą na prostokątnej planszy o wymiarach 10 na 15 jednostek. Plansza reprezentuje pastwisko. Jej górny lewy róg ma współrzędne (0,0). Dodatkowo mapa podzielona jest na jednostkowe kwadraty.

W grze występują jednostki reprezentowane przez okręgi. Okręgi mogą na siebie nachodzić. Są trzy rodzaje jednostek: pasterze, psy pasterskie i owce.

Pasterze i psy są sterowane przez graczy, natomiast owce przez silnik gry. Pasterze i psy poruszają się ze stałą prędkością ustaloną przed rozpoczęciem gry przez silnik. Owce mają trzy prędkości zależne od okoliczności. Owca porusza się szybciej gdy w jej pobliżu znajdują się jednostki gracza. Dodatkowo owca może być przestraszona przez psa, który szczeknie – wtedy porusza się najszybciej, próbując uciec od potencjalnego zagrożenia.

Jeśli pasterz znajduje się dostatecznie blisko owcy (ich okręgi się przecinają) może ją ostrzyc i tym sposobem pozyskać wełnę.

Niektóre pola na planszy to szopy. Każdy z graczy ma do siebie przypisaną jedną szopę, która nie może zostać przejęta. Do przejmowania szop neutralnych wykorzystuje się psy. Gracz z większą ilością psów na polu szopy przejmuje nad nią kontrolę. Gdy obaj gacze posiadają taką samą liczbę psów w szopie, nie zmienia ona właściciela. W momecie zejścia z pola szopy wszystkich psów, staje się ona neutralna.

Pasterze mogą deponować i pobierać z szop wełnę, O wygranej decyduje tylko wełna umieszczona w kontrolowanych przez gracza szopach.

Ruch jednostek

Każdej z jednostek gracza można wydać komendę poruszania się, podając wektor kierunku. Jednostka poruszy się o wektor o tym samym kierunku i zwrocie oraz długości równej prędkości odpowiedniej jednostki. Jednostka znajduje się na polu, w którym zawiera się środek jej okręgu. Jeśli zostanie podany wektor zerowy jednostka nie poruszy się.

W momencie gdy docelowa pozycja jednostki znajduje się poza planszą, jednostka zostanie przesunięta zaraz obok brzegu planszy, a dokładniej współrzędnej poza planszą przypisywana jest wartość równa współrzędnej krawędzi planszy zmniejszonej o 10^{-6}.

Akcje ruchu wykonywane są jako pierwsze.

Strzyżenie

Jeśli okręgi owcy i pasterza przecinają się, można mu wydać komendę strzyżenia. Jeśli wielu pasterzy będzie próbowało zacząć strzyc tę samą owcę, żadnemu z nich się to nie uda. Jeśli jakiś pasterz strzyże już owcę, każda próba rozpoczęcia jej strzyżenia przez innego skończy się niepowodzeniem.

Jeśli akcja strzyżenia się powiedzie, o ile owca posiada wełnę a pasterz nie przekroczył maksymalnej ładowności, jedna jednostka wełny zostanie przekazana pasterzowi. Podczas strzyżenia pasterz i owca są unieruchomieni. Aby kontynuować strzyżenie gracz musi powtórzyć komendę. Podanie jakiejkolwiek innej komendy będzie skutkowało zaprzestaniem strzyżenia.

Szczekanie

Psom można wydać komendę szczekania. Szczekanie będzie skutkowało przestraszeniem owiec znajdujących się w pewnej odległości od szczekającego psa, a następnie ich ucieczką od źródła szczekania przez kilka kolejnych tur. Pies może szczeknąć raz na kilka tur, ich liczba będzie podana na starcie gry.

Szopy

Szopa zajmuje całą komórkę, na której się znajduje. By przeprowadzić interakcję z szopą środek okręgu jednostki musi się znajdować w komórce szopy.

Pasterze mogą zanosić lub wynosić wełnę jedynie z szop których właścicielem jest gracz kontrolujący danego pasterza. Jeśli pasterz spróbuje pobrać z szopy więcej wełny niż się w niej znajduje, pobierze ilość równą wełnie w szopie. Kolejność pasterzy przy pobieraniu wełny jest taka sama jak w wydanych komendach.

Szopy które na początku rozgrywki były neutralne, mogą zmieniać właścicieli. Po przejmowania szop używa się psów. Właścicielem szopy zostaje gracz, który posiada więcej psów na polu szopy. Jeśli pod koniec tury na polu szopy znajdzie się taka sama ilość psów obu graczy przynależność szopy się nie zmieni. Jeśli z pola szopy która ma właściciela znikną wszystkie psy, szopa straci właściciela i stanie się neutralna.

4.2.2. Wejście

Dane inicjalizacyjne

Przed pierwszą turą każdy z graczy dostaje informacje o niezmiennikach w obrębie jednej gry:

- myId – identyfikator gracza,
- initialSheepWool – liczba jednostek wełny, którą posiada każda owca na starcie gry,
- shepardMaxWool – maksymalna ilość wełny jaką może przenosić pojedynczy pasterz,
- entityRadius – promień wszystkich jednostek,
- sheepSpeed1 – prędkość owcy bez modyfikatorów,
- sheepSpeed2 – prędkość owcy, która oddala się od jednostek graczy,
- sheepSpeed3 – prędkość owcy, która została wystraszona przez szczekanie psa,
- shepardSpeed – prędkość pasterza,
- dogSpeed – prędkość psa,
- dangerRadius – odległość, jaką owce będą starały się zachować od jednostek gracza,
- barkCoolDown – liczba rund, które musi odczekać pies pomiędzy szczekięciami,
- barkRadius – odległość psa od owiec wystarczająca, aby je szczekanie wystraszyło,
- calmCoolDown – liczba tur, po których owca przestanie się bać szczekania,
- sheepCnt – liczba wszystkich owiec,
- shepherdsCnt – liczba wszystkich pasterzy,
- dogsCnt – liczba wszystkich psów,
- shedsCnt – liczba wszystkich szop,
- myUnitsCnt – sumaryczna liczba wszystkich posiadanych jednostek.

Dane co turę

Na początku każdej tury gracz otrzymuje następujące informacje:

- opis owiec, na który składa się liczba linii równa liczbie owiec. Każda linia składa się z:
  - id – identyfikator owcy,
  - x; y – pozycja owcy,

  - wool – ilość wełny, jaka znajduje się na owcy,
  - shearedBy – identyfikator pasterza strzygącego owce lub 0, gdy nie jest strzyżona,

- opis wszystkich pasterzy, na który składa się liczba linii równa liczbie pasterzy. Każda linia składa się z:
  - id – identyfikator pasterza,
  - x; y – pozycja pasterza,
  - wool – ilość wełny, jaką niesie pasterz,
  - shearing – identyfikator owcy, którą strzyże pasterz lub 0, gdy nie strzyże żadnej owcy,
  - owner – identyfikator gracza, który jest właścicielem pasterza,

- opis wszystkich psów, na który składa się liczba linii równa liczbie psów. Każda linia składa się z:
  - id – identyfikator psa,
  - x; y – pozycja psa,
  - owner – identyfikator gracza, który jest właścicielem psa,

- opis wszystkich szop, na który składa się liczba linii równa liczbie szop. Każda linia składa się z:
  - x; y – pozycji komórki, na której znajduje się szopa,
  - owner – identyfikator gracza, który jest aktualnym właścicielem szopy,
  - wool – ilość wełny, jaka znajduje się w szopie.

4.2.3. Komendy

Każdej z kontrolowanych jednostek należy wydać jedną z poniższych komend:

MOVE

Jest to komenda ruchu. Można ją wydać psu lub pasterzowi. Komenda przyjmuje trzy argumenty:

- id – numer identyfikacyjny jednostki,
- x, y – składowe wektora kierunku ruchu.

SHEAR

Jest to polecenie strzyżenia owcy. Można je wydać tylko pasterzowi. Komenda przyjmuje dwa argumenty:

- id – numer identyfikacyjny pasterza,
- sheepId – numer identyfikacyjny owcy, która powinna zostać ostrzyżona.

TRANSFER_WOOL

Jest to komenda deponowania lub wybrania wełny z szopy. Można ją wydać tylko pasterzowi który znajduje się na polu z szopą. Przyjmuje trzy argumenty:

- id – numer identyfikacyjny pasterza,
- isDeposit – 1 lub 0, typ operacji (1 – deponowanie, 0 – wybieranie),
- amount – ilość jednostek wełny, która ma być transferowana.

BARK

Jest to polecenie szczekania. Można ją wydać tylko psu. Komenda przyjmuje jeden argument:

- id – numer identyfikacyjny psa.

4.2.4. Opis wizualizacji

Poniżej znajduje się jedna klatka z wizualizacji gry. Widać na niej wszystkie rodzaje jednostek. Tak jak w przypadku gry „Malowanie” kolor żółty i niebieski są przypisane do jednostek gracza. Psy gracza niebieskiego pilnują neutralnych szop, a pasterz goni owcę. Owce które zostały ostrzyżone i nie posiadają już na sobie wełny, mają cieliste korpusy. Tutaj również został uchwycony pasek z podpowiedziami. Podobnie jak przy „Malowaniu”, po prawej od planszy znajduje się pasek aktualnych wyników.

Rysunek 4.2: Przykładowa klatka z wizualizacji „Owiec”

Rozdział 5.

Opis techniczny

5.1. Części wspólne obu gier

Każda z gier korzysta ze wszystkich modułów CodinGame SDK opisanych w rozdziale 3.1 . Obiektowy model projektu, czyli plik **pom.xml** znajdujący się w korzeniu projektu i potrzebny Maven'owi do zaciągnięcia niezbędnych zależności i generowania dokumentacji jest w obu przypadkach niemal taki sam. Jedyną różnicą jest nazwa projektu.

Oba projekty posiadają katalog **config**, w którym znajdują się kolejno pliki:

Boss.java

Jest to przykładowa implementacja prostego bota. Użytkownicy CodinGame mogą z nim konkurować sprawdzając na wizualizacji działania swoich rozwiązań.

config.ini

Plik konfiguracyjny ustalający tytuł gry, jej typ (w tym przypadku „multi" – wieloosobowy) i minimalną oraz maksymalną liczbę graczy która wynosi dwa w obu przypadkach.

statement_en.html

W tym pliku opisane są wszystkie zasady gry w języku angielskim przy użyciu języka HTML.

stub.txt

Tekstowy plik w którym znajduje się psełdokod rozumiany przez **stub generator**. Na jego podstawie generowane są szablony rozwiązań we wszystkich wspieranych językach.

5.1.1. Wizualizacje

W katalogu pod ścieżką **src/main/resources** względem korzenia projektu znajdują się dwa pliki oraz katalog używane do wyświetlania wizualizacji.

demo.js

Przy pierwszym podejściu do każdej gry na CodinGame, gdy użytkownik nie wysłał jeszcze swojego rozwiązania, odgrywana jest przykładowa rozgrywka. W powyzszym pliku znajduje się jej opis klatka po klatce.

config.js

Jest to plik konfiguracyjny do wizualizacji. Określa listę używanych modułów i kolory graczy.

Katalog assets

Pod tym katalogiem znajdują się wszystkie pliki graficzne, które są potrzebne na użytek wizualizacji.

5.1.2. Katalog javadoc

W tym miejscu znajduje się wygenerowana dokumentacja kodu, przy pomocy narzędzia javadoc[18].

5.1.3. Testy

W każdym z projektów znajduje się katalog **test**, w którym jest umieszczona klasa z metodą **main** umożliwiająca uruchomienie programu lokalnie. Znajdują się tam też implementację dwóch botów, które mają ze sobą konkurować w trakcie uruchomienia.

5.1.4. Klasy Raferee i Player

W przypadku obu gier można również wyróżnić przynajmniej dwie klasy z taką samą odpowiedzialnością i nieznacznie różniącą się zaimplementowaną logiką.

Referee

Jest to główna klasa projektu. Jej odpowiedzialnością jest zarządzanie całym cyklem gry. Służy również jako warstwa komunikacyjna pomiędzy modułami. Dziedziczy ona po **AbstractReferee** udostępnionej w CodinGame SDK i nadpisuje (**@Override**) następujące metody:

- init – Wywoływana tylko raz. Inicjuje niezbędne moduły i wywołuje inicjalizację innych klas.
- gameTurn – Metoda ta jest wołana co turę. Zbiera wszystkie komendy od graczy w listę, a następnie przekazuje ją do wykonania klasie **GameState**, której opis znajduje się poniżej. Po rozpatrzeniu wszystkich akcji, wywołuje metody odpowiedzialne za aktualizację stanu wizualizacji.
- onEnd – Funkcja ta jest wykonywana tylko raz, na końcu gry. Aktualizuje ona wyniki i ekran końcowy.

Player

Klasa ta w obu przypadkach posiada pole **pawns** oznaczające liczbę jednostek kontrolowanych przez gracza, która jest równoważna liczbie oczekiwanych komend zwracanych przez metodę **getExpectedOutputLines**. Obie implementacje gier posiadają w tej klasie metodę **getActions**, która to parsuje komendy wypisane przez graczy na wyjście standardowe do obiektów odpowiednich akcji.

5.2. Klasy GameState i Board

W obu projektach znajdują się tytułowe klasy, jednak ich logika znacznie różni. Odpowiedzialnością tych klas jest wykonywanie akcji gracza oraz przechowywanie informacji na temat świata gry i symulowanie zachowań w nim. Szczegóły implementacyjne można znaleźć w dołączonej do pracy dokumentacji.

5.3. Malowanie

5.3.1. Hierarchia

```
.
├── Action.java
├── GameState.java
├── Player.java
├── Referee.java
├── board
│   ├── Board.java
│   ├── Entity.java
│   ├── Pawn.java
│   ├── PawnColors.java
│   └── WallTemplate.java
└── util
```

```
├── LineSeparator.java
├── Util.java
└── Vector2.java
```

5.3.2. Podział na pakiety

W projecie „Malowanie" znajdują się dwa następujące pakiety:

util

Pakiet w którym znajdują się „narzędzia" wykorzystane w implementacji gry. Pośród nich jest klasa **LineSeparator**, która jest własną implementacją metody **lines**, dostępnej od wersji języka Java 11.

board

Klasy znajdujące się w tym pakiecie, używane są do opisu i generowania planszy oraz pionków.

5.3.3. Przykładowy bot

Bot zaimplementowany w tej grze wykonuje dla każdego pionka w cyklu następującą sekwencję:

- pionek przez 4 tury strzela farbą, w każdym kierunku, na odległość pięciu kratek.
- pionek przesuwa się w losowym kierunku. Kierunek w którym znajdują się pionki przeciwnika wylosowany zostaje z większym prawdopodobieństwem niż inne kierunki.

5.4. Owce

5.4.1. Hierarchia

```
.
├── ConstsSettings.java
├── GameState.java
├── Player.java
├── Referee.java
├── actions
│   ├── AbstractAction.java
│   ├── BarkAction.java
```

```
    ├── InvalidAction.java
    ├── MoveAction.java
    ├── ShearAction.java
    └── TransferWoolAction.java
├── board
    ├── Board.java
    ├── Cell.java
    ├── CellType.java
    └── Shed.java
├── entity
    ├── Dog.java
    ├── Entity.java
    ├── Sheep.java
    └── Shepherd.java
└── util
    ├── Util.java
    └── Vector2.javaa
```

5.4.2. Podział na pakiety

W „Owcach" znajdują się cztery pakiety opisane poniżej.

actions

Pakiet ten posiada klasy odpowiadające poszczególnym komendom, które może wydawać gracz. Wszystkie z nich dziedziczą po abstrakcyjnej klasie **AbstractAction**.

board

Znajdują się tutaj klasy odpowiedzialne za opis nieruchomych składników planszy (pastwiska i szop).

entity

W tym pakiecie znajdują się klasy opisujące wszystkie ruchome jednostki w grze.

util

W pakiecie **util** zostały umieszczone klasy „narzędzia" wykorzystane przy implementacji.

5.4.3. Przykładowy bot

W zaproponowanym bocie, psy i pasterze są sterowane niezależnie od siebie. Psy starają się przejąć jak największą liczbę szop, które nie mają właściciela. Komenda **BARK** nie jest wykorzystywana. W przypadku gdy na planszy nie ma dodatkowych szop, psy pozostają w bezruchu przez całą grę.

Jedynym zadaniem pasterzy jest podbieganie do najbliższej, jeszcze nieogolonej owcy i strzyżenie jej. Następnie po skończeniu strzyżenia pasterz kieruje się do najbliższej posiadanej przez bota szopy i deponuje tam całą wełnę. Gdy wełna zostanie odłożona, kieruje się on do kolejnej owcy. Taki cykl powtarzany jest dla każdego pasterza do końca gry.

Rozdział 6.

Podsumowanie

Praca miała na celu wykorzystanie ciekawych gier pochodzących z zawodów PIZZA i udostępnienie ich szerszej publiczności, w miejscu gdzie każdy może się z nimi zmierzyć. Wybór gier ze względów technicznych oraz dzięki ich nietypowości, padł na „Malowanie" i „Owce".

Gry zaimplementowane na platformę CodinGame można znaleźć pod tymi odnośnikami:

- Malowanie [1]
- Owce [2]

Praca spełni swój cel, jeśli na platformie CodinGame pojawi się wiele nowych pomysłów na boty do tych gier. Dla osób zainteresowanych pozostałymi zadaniami z zawodów PIZZA, ich treści można znaleźć tutaj[19].

W ramach dalszego rozwoju pracy możliwa jest implementacja kolejnych gier, usprawnienie przykładowych botów lub granulacja już zaimplementowanych gier na ligi, co ułatwi nowym odbiorcom zrozumienie zasad gry i jest umożliwiane przez platformę CodinGame.

[1] `https://www.codingame.com/contribute/view/29286566c75fa9e336b39cc5b4fecf903ef40`
[2] `https://www.codingame.com/contribute/view/29352e0afabdab8a19126f5d9bce6c05f57ca`

Bibliografia

[1] PIZZA. Strona główna zawodów pizza. https://contest.pizza/index.html.

[2] CodinGame. Strona główna platformy codeingame. https://www.codingame.com/home.

[3] kubernetes. Podstawowe informacje o kubernetesie. https://kubernetes.io/.

[4] PIZZA. Oryginalna treść gry „malowanie". https://contest.pizza/static/tasks/2016/finals/malowanie.8c707ed678e8.pdf.

[5] PIZZA. Oryginalna treść gry „owce". https://contest.pizza/static/tasks/2017/finals/owce.0519f50928ea.pdf.

[6] CodinGame. Lista wspieranych przez codingame języków. https://www.codingame.com/playgrounds/40701/help-center/languages-versions.

[7] Oracle. Dokumentacja jdk 8. https://devdocs.io/openjdk~8/.

[8] CodinGame. Materiały twórców codingame na platformie github. https://github.com/CodinGame.

[9] CodinGame. Dokumentacja modułu core. https://www.codingame.com/playgrounds/25775/codingame-sdk-documentation/game-manager.

[10] CodinGame. Dokumentacja modułu entities. https://www.codingame.com/playgrounds/25775/codingame-sdk-documentation/introduction-3.

[11] CodinGame. Dokumentacja modułu runner. https://www.codingame.com/playgrounds/25775/codingame-sdk-documentation/game-runner.

[12] CodinGame. Dokumentacja modułu tooltip. https://www.codingame.com/playgrounds/25775/codingame-sdk-documentation/tooltip-module.

[13] CodinGame. Dokumentacja modułu endscreen. https://www.codingame.com/playgrounds/25775/codingame-sdk-documentation/tooltip-module.

[14] Apache. Wprowadzenie do narzędzia maven. https://maven.apache.org/what-is-maven.html.

[15] Google. Dokumentacja google guice. https://github.com/google/guice.

[16] PixiJS. Dokumentacja pixijs. https://pixijs.download/release/docs/index.html.

[17] CodinGame. Dokumentacja stub generator. https://www.codingame.com/playgrounds/40701/help-center/stub-generator.

[18] Oracle. Informacje na temat javadoc. https://www.oracle.com/java/technologies/javase/javadoc-tool.html.

[19] PIZZA. Wszystkie treści zadań z zawodów pizza. https://contest.pizza/tasks/.