

## Baza Danych

W projekcie nie ma żadnych zapisanych użytkowników, a dostęp do bazy jest przewidziany tylko poprzez serwer w konkretnych przypadkach przy konkretnych danych. Postanowiono więc zostawić domyślną regułę bezpieczeństwa - blokuj wszystko - dostęp do bazy ma tylko serwer poprzez AdminSDK. Aby serwer mógł dostać się do bazy danych potrzebne jest połączenie z Internetem. Każda kolekcja jest osobnym pokojem rozgrywki. Jego nazwa to losowo wygenerowane 20 znaków (litery + cyfry). Natomiast w każdym pokoju są 3 dokumenty:

1. General - znajdują się tam informacje potrzebne obu graczom;
2. Player1 - informacje o stanie pierwszego gracza;
3. Player2 - informacje o stanie drugiego gracza.

W dokumencie General znajdują się pola:

- day - pole typu number - przyjmuje wartości od 1 do  $\infty$  - określa numer tury, wykorzystywane jest przy sprawdzaniu gotowości przejścia do następnej tury;
- field - pole typu number - przyjmuje wartości od 0 do 6 - określa między którymi polami aktualnie jest toczonej bitwa, wykorzystywane przy obliczeniach dotyczących aktualnej bitwy oraz przy wysyłaniu informacji o wrogich jednostkach do gracza na początku każdej tury;
- no\_players\_ready - pole typu number - przyjmuje wartości od 0 do 2 - określa ilu graczy ukończyło aktualną turę, wykorzystywane jest przy sprawdzaniu gotowości obliczenia następnej tury;
- tot - pole typu number - przyjmuje wartości od 0 do 5 - określa przez ile tur nie będzie toczonej bitwa, wykorzystywane przy obliczeniach dotyczących aktualnej bitwy;
- war - pole typu boolean - określa czy będzie aktualnie prowadzona bitwa, wykorzystywane razem i zależne od pola tot, gdy ono przyjmuje wartość 0 war jest ustawione na true, w przeciwnym wypadku false;
- winner - pole typu number - określa w danej rozgrywce wyłonił się zwycięzca czy jeszcze nie, wykorzystywane i ustawiane przy sprawdzaniu zwycięzcy po każdym obliczaniu tury.

Struktura dokumentów Player1 i Player2 jest identyczna. Oba te dokumenty zawierają takie same pola, a każdy z graczy przechowuje w odpowiednim informacje o stanie swojej rozgrywki. Jeden gracz nie powinien widzieć co przechowuję drugi

(wyjątek stanowi jedno pole, które opiszę niżej). W dokumentach Player1 i Player2 znajdują się następujące pola:

- gold - pole typu number - określa ile złota miał pod koniec poprzedniej turze dany gracz. Pole zostało stworzone w ramach późniejszego rozwoju gry, gdyż aktualnie nie jest ono wykorzystywane;
- tech - pole typu number - określa ile punktów technologii miał pod koniec poprzedniej turze dany gracz. Pole również zostało stworzone w ramach późniejszego rozwoju gry, gdyż też nie jest aktualnie wykorzystywane;
- unit\_stats - pole typu map - zawiera w sobie szereg pól opisujących statystyki jednostek danego gracza. W polu tym znajdują się takie pola jak:
  - archer\_armor - pole typu number - określa ile punktów pancerza mają jednostki łucznicze gracza;
  - archer\_damage\_min - pole typu number - określa minimalną wartość obrażeń zadawanych przez jednostki łucznicze;
  - archer\_damage\_max - pole typu number - określa maksymalną wartość obrażeń zadawanych przez jednostki łucznicze;
  - archer\_speed - pole typu number - określa prędkość jednostek łuczniczych. Aktualnie pole nie jest używane - szczegółowy opis jak ma być ono używane znajduje się w podrozdziale "Rozwój Gry";
  - cavalry\_armor - pole typu number - określa ile punktów pancerza mają jednostki konne gracza;
  - cavalry\_damage\_min - pole typu number - określa minimalną wartość obrażeń zadawanych przez jednostki konne;
  - cavalry\_damage\_max - pole typu number - określa maksymalną wartość obrażeń zadawanych przez jednostki konne;
  - cavalry\_speed - pole typu number - określa prędkość jednostek konnych. Aktualnie pole nie jest używane - szczegółowy opis jak ma być ono używane znajduje się w podrozdziale "Rozwój Gry";
  - footman\_armor - pole typu number - określa ile punktów pancerza mają jednostki piechoty gracza;
  - footman\_damage\_min - pole typu number - określa minimalną wartość obrażeń zadawanych przez jednostki piechoty;
  - footman\_damage\_max - pole typu number - określa maksymalną wartość obrażeń zadawanych przez jednostki piechoty;

- footman\_speed - pole typu number - określa prędkość jednostek piechoty. Aktualnie pole nie jest używane - szczegółowy opis jak ma być ono używane znajduje się w podrozdziale "Rozwój Gry".
  - units - pole typu map - zawiera w sobie informacje o rozmieszczeniu jednostek gracza w postaci list 3-elementowych (typu number) nazwanych od '0' do '7'. Każda taka lista reprezentuje liczbę wojsk danego gracza na danym polu. Pierwszą wartością tej listy jest liczba wojsk piechoty, drugą jest liczba wojsk łuczniczych, a ostatnią jest liczba wojsk kawalerii gracza. Serwer dba aby aby wartości w tych listach dla dokumentu Player1 i Player2 wzajemnie się wykluczały tzn. jeżeli jakiś gracz posiada jakieś wojsko w liście o numerze n to w liście drugiego gracza pod tym numerem jest [0,0,0];
  - upgades - pole typu map - zawiera w sobie informacje o poziomach ulepszeń dla danego gracza. W polu tym znajdują się kolejne pola:
    - arch\_att - pole typu number - określa poziom ulepszenia ataku łuczników;
    - arch\_def - pole typu number - określa poziom ulepszenia pancerza łuczników;
    - cav\_att - pole typu number - określa poziom ulepszenia ataku kawalerii;
    - cav\_def - pole typu number - określa poziom ulepszenia pancerza kawalerii;
    - foot\_att - pole typu number - określa poziom ulepszenia ataku piechoty;
    - foot\_def - pole typu number - określa poziom ulepszenia pancerza piechoty;
    - gold - pole typu number - określa poziom ulepszenia przyrostu złota;
    - tech - pole typu number - określa poziom ulepszenia przyrostu punktów technologii;
- Każde z tych pól jest wykorzystywane podczas obliczeń końca turы - gdy podnoszone są statystyki jednostek dostępne w polu unit\_stats oraz przyznawaniu dodatkowego złota i punktów technologii na początku każdej turы gracza.

## Serwer

Podczas gdy baza danych przechowuje informacje o danej grze, to serwer przetwarza wszystkie dane oraz zajmuje się komunikacją między użytkownikami oraz bazą danych. Jest on napisany w języku Python oraz do komunikacji z bazą danych wykorzystuje bibliotekę firebase-admin, a do komunikacji z użytkownikami bibliotekę gRPC. Z dodatkowych używanych bibliotek które wymagają instalacji jest numpy (wykorzystywany w jednej metodzie do modyfikowania list z jednostkami), oraz threading, a z takich które są dostępne bez instalacji używane są time oraz json.

### Konfiguracja

Plik który jest potrzebny przed uruchomieniem serwera to config.txt. Jego zadanie jest bardzo proste - przechowuje konfigurację niezbędną podczas uruchamiania serwera. Znajduje się w nim JSON pokazany na listingu 15, który zawiera w sobie informacje:

```
{ "ip_address" : "...",
  "port" : "9999",
  "threads" : 10,
  "base_driver" : "./mg-db-4251d.firebaseio-adminsdk-qdwkt-92102e2f75.json"}
```

*Listing 15 - plik config.txt*

gdzie:

- ip\_address - adres IP na którym ma nasłuchiwać serwer;
- port - port na którym ma nasłuchiwać serwer;
- threads - liczba wątków na których ma serwer działać;
- base\_driver - ścieżka i nazwa klucza prywatnego bazy danych.

**Uwaga!: Aby uruchomić serwer należy stworzyć własną bazę Firebase oraz udostępnić serwerowi jej klucz prywatny.**

## Budowa Serwera

Serwer został napisany w sposób obiektowy i składa się z 8 głównych plików oraz 2 pomocniczych.

Plik .proto wykorzystywany w projekcie zawiera 5 komunikatów. Są nimi:

- JoinMatchmaking - wysyła wiadomość New\_game, a zwraca wiadomość Game - wykorzystywany podczas parowania dwóch oczekujących graczy;

- Matchmaking\_Ready - wysyła wiadomość New\_game\_status, a zwraca wiadomość Status - wykorzystywany przez gracza w celu sprawdzenia czy znaleziono mu już przeciwnika;
- End\_Turn - wysyła wiadomość Forward\_to\_server, a zwraca wiadomość Status - wykorzystywany do wysyłania informacji o wykonanej przez gracza turze;
- Next\_Turn\_Status - wysyła wiadomość Game\_status, a zwraca wiadomość Status - wykorzystywany do sprawdzenia przez gracza czy serwer już przetworzył nową turę;
- Next\_Turn - wysyła wiadomość Next\_Turn\_Info, a zwraca wiadomość Forward\_to\_player - wykorzystywany przez gracza do pobrania informacji o nowej turze.

Każdy z tych komunikatów jest unarny - pojedyncza wiadomość jest wysyłana oraz pojedyncza jest odpowiedzią. Kolejnymi elementami w pliku .proto są wiadomości. Ich zawartość jest następująca:

- New\_game - zawiera w sobie pojedyncze pole int32 - name - aktualnie zawsze przyjmuje wartość 0. W przyszłości po dodaniu opcji dołączania do konkretnych gier będzie ono wykorzystywane właśnie do dołączenia do owego pokoju. Wiadomość ta jest wysyłana w momencie gdy użytkownik prosi serwer o znalezienie mu przeciwnika;
- Game - zawiera w sobie dwa pola int32 oraz jedno string. Pierwszym z nich jest int32 stat - pole to określa czy użytkownik dołączył się do już istniejącego pokoju czy też należało stworzyć mu nowy. Przyjmuje wartość 1 w przypadku dołączenia do istniejącego oraz 0 w przypadku nowego. Drugim polem jest string id - jest to id pokoju zwrócone przez bazę danych. Trzecie pole to int32 uid - zawiera w sobie informacje czy gracz jest graczem 1 czy 2;
- New\_game\_status - zawiera w sobie pojedyncze pole string - name. Pole to zawiera w sobie to co jest zwracane przez wiadomość Game w polu id - id pokoju gry. Wiadomość ta jest wysyłana tylko wtedy gdy pole stat z wiadomości Game wynosić będzie 0;
- Status - zawiera w sobie pojedyncze pole int32 - stat. Wykorzystywane w 3 różnych komunikatach jako potwierdzenie że wiadomość dostarczona pomyślnie lub że coś jest gotowe (można pobrać nową turę, znaleziono przeciwnika);

- Forward\_to\_server - zawiera w sobie 14 pól int32 oraz jedno string. Wysyła informacje o ruchach gracza w aktualnej turze. Pole tekstowe jest to id pokoju do którego dane mają zostać zapisane. Trzy pola int32 - foot, arch i cav - oznaczają liczbę wojsk rekrutowanych przez gracza. Pole player określa czy użytkownik jest graczem 1 lub 2. Osiem pól int32 - gold\_up, tech\_up, up\_foot\_att, up\_foot\_def, up\_arch\_att, up\_arch\_def, up\_cav\_att, up\_cav\_def - przechowują informacje czy gracz dokonał konkretnego ulepszenia - kolejno: złota, punktów technologii, ataku piechoty, obrony piechoty, ataku łuczników, obrony łuczników, ataku kawalerii, obrony kawalerii. Dwa pola int32 - gold i tech - wysyłają informacje o pozostałym złocie i punktach technologii gracza;
- Game\_status - zawiera jedno pole string i jedno int32. Pole string gid jest polem określającym id pokoju gry, pole int32 day określa aktualną turę u użytkownika. Wiadomość ta jest wysyłana w celu sprawdzenia czy serwer przetworzył już aktualną turę (czy dwóch graczy wysłało wiadomość forward\_to\_serwer i czy serwer przetworzył te informacje);
- Next\_Turn\_Info - zawiera w sobie pojedyncze pole string oraz int32. Pole string gid określa id pokoju gry, pole int32 player, który z graczy prosi o dane. Wiadomość ta jest wysyłana tylko w przypadku gdy wiadomość Status w komunikacie Next\_Turn\_Status zwróciła 1;
- Forward\_to\_player - zawiera w sobie 14 pól int32, dwie listy int32 oraz jedno pole typu bool - zwraca użytkownikowi informacje o przetworzonej przez serwer nowej turze. Listy - player, enemy - zawierają w sobie informacje o rozmieszczeniu jednostek gracza i przeciwnika (gracza na każdym posiadanym polu, przeciwnika na polu spornym). Pole bool war - zawiera informacje o tym czy aktualnie będzie toczona walka czy też jest chwilowe zawieszenie broni. Trzy pola int32 - day, field i winner - zawierają informacje o numerze aktualnej turzy, które pole jest aktualnie sporne oraz czy jest już zwycięzca gry. Dwa pola int32 - gold i tech - opisują fundusze gracza na aktualną turę (złoto i punkty technologii). Dziewięć pól int32 - foot\_att\_min, foot\_att\_max, foot\_armor, arch\_att\_min, arch\_att\_max, arch\_armor, cav\_att\_min, cav\_att\_max, cav\_armor - zawierają statystyki poszczególne statystyki poszczególnych jednostek - minimalną wartość ataku, maksymalną wartość ataku oraz pancerz piechoty, łuczników i kawalerii.

Aby uruchomić serwer należy użyć komendy:

```
$ python ./server.py lub $ python3 ./server.py.
```

Jeżeli serwer uruchomił się poprawnie to w konsoli będą wyświetlane komunikaty o ilości wątków wykorzystywanych aktualnie przez serwer (server active: on threads).

Aby zatrzymać serwer należy wcisnąć CTRL + C. Wyświetli się wtedy komunikat keyboard interrupt.

Podczas uruchamiania serwera pierwszą zrobioną przez niego rzeczą jest utworzenie pustej listy free\_games - odpowiedzialnej za przechowywanie nazw pokoi w których oczekuje gracz na swojego przeciwnika, następnie tworzona jest klasa z pliku base.py - Base() - odpowiedzialna za komunikację z bazą danych. Po tym tworzony jest sam serwer - wykonywana jest funkcja server(), która widoczna jest na listingu 16.

```
def server():
    configs = '[' + info['ip_address'] + ']:' + info['port']
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=info['threads']))
    game_service_pb2_grpc.add_TTTServicer_to_server(Listener(), server)
    server.add_insecure_port(configs)
    server.start()
    try:
        while True:
            print("server active: on threads %i" % (threading.active_count()))
            time.sleep(10)
    except KeyboardInterrupt:
        print("Keyboard interrupt")
        server.stop(0)
```

*Listing 16 - funkcja server() z pliku server.py*

Funkcja ta ustawia maksymalną wartość wątków, a następnie binduje adres ip oraz port. W międzyczasie tworzona jest klasa Listener() - odpowiedzialna za komunikację serwera z użytkownikami. Po wykonaniu tego wszystkiego serwer oczekuje w nieskończonej pętli na żądania od graczy. Klasa Listener() jest ściśle związana z komunikatami z pliku .proto - to w tej klasie definiowany jest sposób działania tych komunikatów. Podczas jej tworzenia przypisywana jest do niej klasa Base (stworzona wcześniej) oraz tworzony jest semafor. Metody klasy Listener() są nazwane dokładnie jak nazwy komunikatów z pliku .proto, a ich działanie jest następujące:

- JoinMatchmaking - metoda odpowiedzialna za sparowanie graczy - korzystając z semafora, sprawdza czy w liście free\_games znajduje się

element. Jeżeli tak to usuwa z niej pierwszy element oraz uruchamia metodę add\_to\_game z klasy Base() po czym zwraca użytkownikowi wiadomość Game (ze statusem 1), jeżeli nie to uruchamia metodę create\_game z klasy Base() po czym dodaje do listy id zwrócone przez tą metodę. Na koniec zwraca użytkownikowi wiadomość Game (ze statusem 0);

- Matchmaking\_Ready - odpowiada za sprawdzenie czy przesłana przez użytkownika nazwa gry znajduje się jeszcze na liście gier oczekujących. Jeżeli tak to zwraca komunikat Status - stat = 0, a jeżeli nie to stat = 1;
- End\_Turn - uruchamia metodę update\_player z klasy Base() w celu zapisania przesłanych od użytkownika informacji o aktualnie wykonanej turze - zwraca wiadomość Status z polem stat o wartości zwracanej przez metodę update;
- Next\_Turn\_Status - uruchamia metodę player\_turn\_status z klasy Base() w celu sprawdzenia czy rozpoczęła się już nowa tura. Zwraca wiadomość Status z polem stat o wartości zwracanej przez metodę player\_turn\_status;
- Next\_Turn - uruchamia metodę send\_next\_turn z klasy Base() w celu przesłania graczowi informacji o nowej turze. Przekazuje do niego odpowiedź zwróconą przez metodę send\_next\_turn.

```
def __init__(self, file):
    self.cred = credentials.Certificate(file)
    self.default_app = firebase_admin.initialize_app(self.cred)
    self.db = firestore.client()
    self.battle = Battle()
    self.cost = Cost()
    self.unit_stat = Stats()
```

Listing 17 - konstruktor klasy Base()

Konstruktor klasy Base() z pliku base.py - widoczny na listingu 17 - importuje wskazany w pliku config.txt klucz prywatny i tworzy połączenie z bazą danych wykorzystując bibliotekę firebase\_admin. Oprócz tego tworzy 3 nowe klasy:

- Battle() - z pliku battle.py;
- Cost() - z pliku defines.py;
- Stats() z pliku defines.py.

Klasa Base() posiada 6 metod. Są nimi:

- create\_game - metoda używana do tworzenia nowych kolekcji i dokumentów w bazie danych. Na początku losuje 20 znaków (cyfry i litery), które to będą nazwą nowej kolekcji. Następnie sprawdzane jest czy aby wylosowana nazwa nie jest już używana. Jeżeli tak to generowana jest nowa nazwa do momentu gdy w bazie danych nie będzie kolekcji o takiej nazwie. Dalej tworzone są dokumenty General i Player1 jako pythonowe słowniki (Dictionary). W tych słownikach odpowiednim kluczom przypisywane są domyślne wartości. Część z nich znajduje się w klasie Stats():
  - Liczba złota i punktów technologii ustawiane są na 300;
  - Liczba jednostek we wszystkich listach w polu units to zawsze 0;
  - Poziomy wszystkich ulepszeń w polu upgrades to 0;
  - Startowy numer tury to 1;
  - Flaga wojny ustawiana jest na false wraz z 5 turowym zawieszeniem broni;
  - Początkowy numer spornego terytorium to 3;
  - Brak zwycięzcy (winner = 0).

Następnie te słowniki są wysyłane do bazy danych, a metoda zwraca wygenerowane id gry.

- add\_to\_game - metoda używana w parze z create\_game. Jej zadaniem stworzenie dokumentu Player2 w kolekcji stworzonej przez create\_game. Dokument ten ma identyczną strukturę co dokument Player1 tworzony przez metodę create\_game;
- update\_game - metoda używana do utworzenia chwilowego pola next w odpowiednim dokumencie (Player1 lub Player2). Pole to zawiera w sobie następujące informacje:
  - Liczba punktów technologii i złota pod koniec tury danego gracza;
  - Które ulepszenie gracz wynajduje;
  - Jakie jednostki gracz rekrutuje.

Pole Next jest używane podczas wykonywania kalkulacji końca tury. Gdy obaj gracze w pokoju uruchomią tę metodę to wtedy uruchamiana jest metoda next\_turn\_calc oraz pole next jest usuwane;

- player\_turn\_status - metoda wysyła graczowi informacje czy metoda next\_turn\_calc została użyta;
- send\_next\_turn - metoda wysyła graczowi informacje o stanie gry zwróconym przez metodę next\_turn\_calc. Tymi informacjami są:
  - Numer nowej tury;
  - Numer nowego pola spornego;
  - Czy aktualnie będzie prowadzona walka;
  - Nowy stan złota i punktów technologii gracza;
  - Nowe statystyki jednostek - minimalne/maksymalne obrażenia i liczba punktów pancerza dla jednostek piechoty, łuczników i kawalerii;
  - Czy aktualnie jest zwycięzcą;
  - Liczba jednostek gracza na każdym polu;
  - Liczba jednostek wroga na polu spornym.

U każdego gracza jego wróg zawsze wyświetlany jest z lewej strony ekranu, a jego wojsko porusza się od prawej do lewej strony ekranu. Dlatego metoda ta musi odpowiednio "lustrzanie odbić" informacje o numerze aktualnego pola spornego oraz rozmieszczeniu jednostek w przypadku gdy wysyła te informacje do jednego z graczy.

- next\_turn\_calc - jest to najważniejsza metoda z klasy Base(). Gdy oboje graczy zakończy swoje tury wtedy metoda ta:
  1. Zwiększa numer tury;
  2. Sprawdza czy zakończył się okres zawieszenia broni;
  3. Nakłada ulepszenia wynajdowane przez graczy przy pomocy metody update\_tech z klasy Battle();
  4. W przypadku gdy aktualnie nie trwa zawieszenie broni:
    - a. Oblicza wynik bitwy między jednostkami gracza między polami spornymi za pomocą metody battle z klasy Battle();
    - b. Sprawdza czy któryś z graczy nie wygrał aktualnej bitwy. W takim przypadku zmienia flagę zawieszenia broni na True oraz nastawia go na 5 tur;

- c. Sprawdza czy po bitwie któryś z graczy nie zajął już wszystkich pól przez co zwyciężył;
- 5. Przesuwa jednostki gracza o jedno pole w odpowiednią stronę przy pomocy metody move \_units z pliku Battle();
- 6. Aktualizuje informacje z punktów 1-5 w bazie danych.

W pliku `battle.py` znajduje się klasa `Battle()`. Jest ona wykorzystywana tylko w metodzie `next_turn_calc` w klasie `Base()`. Ma ona 3 zadania: przesunąć jednostki, nałożyć ulepszenia i obliczyć wynik bitwy. Do tych celów używa metod:

- `update_tech` - metoda nakłada ulepszenia obu graczom. W tym celu pobiera z bazy danych aktualne poziomy ulepszeń oraz statystyki jednostek. Następnie ulepsza statystyki każdej jednostki (punkty pancerza, minimalna wartość ataku, maksymalna wartość ataku) w następujący sposób
- $$\text{new} = \text{old} + (\text{perLevelStat} * \text{currentLevel} * \text{upgradeStatus}), \text{ gdzie}$$
- `old` - aktualna statystyka jednostki;
  - `perLevelStat` - stała z klasy `Stats()` z pliku `defines.py`;
  - `current_level` - aktualny poziom ulepszenia danej statystyki;
  - `upgradeStatus` - wartość 0/1 określająca czy dany gracz dokonuje konkretnego ulepszenia. W przypadku gdy tego nie robi wartość nowej statystyki wyniesie dokładnie tyle co stara.

Na samym końcu zwiększa poziomy wynajdywanych ulepszeń o 1 w górę oraz zwraca słownik z nowymi wartościami;

- `move units` - metoda przesuwa jednostki obu graczy. Tworzone są 2 słowniki oraz wykonywane dwie pętle (po jednym dla każdego gracza). W przypadku gdy pola są we władaniu gracza przesuwa każdą jednostkę na pole obok. W przypadku gdy pole kontroluje drugi gracz to ustawia wartości jednostek na [0,0,0]. Na koniec zwraca słowniki z nowymi wartościami;
- `battle` - jedna z dwóch metod wykorzystywanych do obliczania wyniku bitwy. Pobiera statystyki jednostek kawalerii każdego gracza, a następnie losuje wartość ich ataku (losowanie liczby pomiędzy minimalną, a maksymalną wartością ataku). Następnie wykonywana jest metoda `fight` z tej samej klasy. Później wykonywane są te same czynności dla jednostek łuczniczych, a na końcu dla piechoty. Na samym końcu sprawdzane jest czy którymuś z graczy

zginęły wszystkie jednostki piechoty. Jeżeli tak to wynik starcia jest ustawiany na porażkę dla tego gracza. Jeżeli oboje graczy straciło wszystkie jednostki piechoty lub każdemu graczowi piechota przeżyła wynik starcia ustawiony jest na remis. Metoda zwraca wynik starca oraz liczbę jednostek każdego typu które przeżyły dla każdego z graczy;

- fight - druga metoda wykorzystywana do obliczania wyniku bitwy.

Wykonywana osobno dla jednostek kawalerii, łuczników i piechoty. Przebieg działania funkcji wygląda następująco:

1. Zadawane są obrażenia tzw. overkill (obrażenia nadmiarowe jednostek z poprzedniego wywołania metody fight podczas działania metody battle - przykładowo jeżeli zdrowie jednostek gracza 1 wynosiło 100, a gracz 2 zadaje 300 obrażeń to jego overkill wynosi 200). Overkill w przypadku pierwszego wywołania metody wynosi 0;
2. Obrażenia jednostek gracza mnożone są przez liczbę jego jednostek, a następnie od tej wartości odejmowana jest wartość pancerza jednostek wroga przemnożona przez liczbę jednostek wroga (dla każdego gracza osobno);
3. Obliczane są nowe łączne punkty zdrowia jednostek (osobno dla każdego gracza) w następujący sposób:

$$p1HP_{new} = (p1HP * p1) - p2Damage, \text{ gdzie}$$

- a.  $p1HP_{new}$  - nowe łączne zdrowie jednostek gracza 1;
- b.  $p1HP$  - aktualne łączne zdrowie jednostek gracza 1;
- c.  $p1$  - liczba jednostek gracza 1;
- d.  $p2Damage$  - obrażenia jednostek gracza 2;

Dla obliczeń gracza 2, indeksy są odwrotne;

4. Obliczany jest overkill każdego gracza;
5. Obliczana jest liczba jednostek które przeżyły bitwę. Wykonywana jest funkcja ceil dostępna w bibliotece math na  $p1HP_{new} / p1HP$ .
6. Zwracana jest liczba jednostek każdego gracza które przeżyły oraz ich overkill.

Ostatnim plikiem serwera jest plik `defines.py` przechowuje w sobie 2 klasy: `Cost()` i `Stats()` których fragmenty są widoczne na listingach 18 i 19

```

class Stats:
    def __init__(self):
        #footman stats
        self.footman_hp = 10
        self.footman_armor = 0
        self.footman_damage_min = 4
        self.footman_damage_max = 4
        self.footman_speed = 0
        self.footman_damage_min_pl = 4
        self.footman_damage_max_pl = 6
        self.footman_armor_pl = 5

```

*Listing 18 - klasa Stats()*

```

class Cost:
    def __init__(self):
        self.cost_gold_tech = 100
        self.cost_tech_gold = 100
        self.cost_foot_gold = 10
        self.cost_foot_att_tech = 100
        self.cost_foot_att_gold = 50
        self.cost_foot_def_tech = 100
        self.cost_foot_def_gold = 50

```

*Listing 19 - klasa Costs()*

Nie mają one w sobie żadnych metod. Ich zadaniem jest przechowywanie stałych, które są wykorzystywane podczas działania serwera i gry.

- Klasa Costs() przechowuje w sobie informacje o kosztach w złocie i punktach technologii które gracz musi zapłacić aby rekrutować wojsko i opracowywać ulepszenia. Serwer wykorzystuje te stałe w metodzie send\_next\_turn z klasy Base() podczas przydzielania graczom przychodu w złocie i punktach technologii.
- Klasa Stats() zawiera stałe o statystykach każdej jednostki oraz jak one się zwiększały wraz z każdym nowym poziomem ulepszeń. Serwer wykorzystuje te dane w metodzie update\_tech z klasy Battle() podczas nakładania ulepszeń.

Plik `client_test.py` służy do “obsługi” drugiego gracza w trybie testowym.

Uruchamia się go za pomocą `$ python ./client_test.py [tryb] [ilość]`  
`[tryb]` przyjmuje 3 wartości:

- 0 - sesja testowa jest resetowana do ustawień początkowych. Sesja musi być resetowana za każdym razem gdy użytkownik uruchamia grę.
- 1 - drugi gracz kończy turę i nie wystawia żadnych jednostek
- 2 - drugi gracz kończy turę i wystawia [ilość] jednostek.

## Architektura Gry

BTI został napisany z wykorzystaniem biblioteki PyGame. Pomocniczymi bibliotekami używanymi w grze są time oraz json.

### Konfiguracja

Plikiem odpowiedzialnym za przetrzymywaniem konfiguracji jest config.txt

```
{  
    "ip_address" : "localhost",  
    "port" : "9999",  
    "test_mode" : 1  
}
```

*Listing 20 - plik config.txt*

Zawarty w nim JSON widoczny na listingu 20 zawiera takie informacje jak:

- adres ip na którym nasłuchuje serwer - ip\_address;
- port na którym nasłuchuje serwer - port;
- włączony tryb gry - test\_mode - 0 w przypadku trybu normalnego lub 1 w przypadku trybu testowego.

### Budowa Gry

W odróżnieniu od serwera, klient gry ma tylko 1 plik w którym znajduje się klasa, która ma metody (nie licząc klas wygenerowanych przez gRPC). Wspomaga ją pięć plików w których znajdują się klasy przechowywujące zmienne i stałe oraz plik tekstowy przechowujący JSON odpowiedzialny za konfigurację gry. Plik .proto oraz 2 pliki pb2 są dokładnie tymi samymi plikami, których używa serwer, dlatego ważne jest aby nie zapomnieć przenieść ich z serwera do gry w przypadku gdy dokona się w nich jakiejkolwiek modyfikacji. W folderze z grą znajduje się folder resources - przechowywane są w nim wszystkie assety (Asset - “Zawartość gotowa do umieszczenia w projekcie przez deweloperów. Assetami mogą być zarówno elementy graficzne, jak i elementy interfejsu, efekty dźwiękowe bądź fragmenty kodu odpowiadające na przykład za AI lub fizykę.”[7]). Każdy asset jest wczytywany przez klasę GFX() z pliku assets.py. Jest ona odpowiedzialna za:

- Przetrzymywanie oraz skalowanie wszelkich plików z grafikami;
- Przetrzymywanie koordynatów x i y każdej ikony;

- Otwieranie i przechowywanie w słowniku danych z pliku desc.txt, który jest odpowiedzialny za przechowywanie opisów.

Plik defines.py który przechowuje klasy Cost() i Stats() jest plikiem identycznym jak plik o takiej samej nazwie na serwerze. W tym przypadku obie klasy są wykorzystywane, aby poprawnie wyświetlać opisy z pliku desc.txt którego fragment jest widoczny na listingu 21.

```
"footman" : {
    "name" : "Train Footman",
    "desc" : [
        "Class: Infantry",
        "Damage: ",
        "Armor: ",
        "HP: "
    ],
}
```

*Listing 21 - przykładowy opis z pliku desc.txt*

Klasa Cost() jest dodatkowo używana w celu poprawnego sprawdzenia czy gracz ma odpowiednią ilość środków na zrekrutowanie wojska oraz opracowanie ulepszenia. Tworzona jest również dodatkowa instancja klasy Stats() w klasie Board() - w tym przypadku przechowuje ona nie stałe, a zmienne które są odpowiedzialne za aktualne statystyki jednostek gracza.

Plik board.py przechowuje klasę Board() - widoczną na listingu 22.

```
class Board:
    def __init__(self):
        self.gid = 0
        self.player = 0
        self.day = 1
        self.war = False
        self.gold = 300
        self.tech = 300
        self.field = 3
        self.tech_level = 1
        self.gold_level = 1
        self.foot_att_level = 1
        self.foot_def_level = 1
        self.arch_att_level = 1
        self.arch_def_level = 1
        self.cav_att_level = 1
        self.cav_def_level = 1
        self.enemy_units = [0,0,0]
        self.player_units = [[0 for col in range(3)] for row in range(8)]
        self.stats = Stats()
```

*Listing 22 - klasa Board()*

Klasa ta odpowiedzialna jest za przechowywanie wszystkich zmiennych, które obsługują aktualną rozgrywkę tzn.:

- ID pokoju gry oraz ID gracza (czy gracz to Player1 czy Player2);
- Stan złota i punktów technologii;
- Numer tury i czy aktualnie toczyć będzie się bitwa;
- Poziomy każdego ulepszenia opracowanego przez gracza;
- Liczbę wojsk wroga na polu spornym;
- Liczbę wojsk gracza na każdym polu;
- Statystyki każdej jednostki gracza (klasa Stats()).

Ostatnią pomocniczą klasą jest Forward() z pliku forward.py, jest ona widoczna na listingu 23.

```
class Forward:  
    def __init__(self):  
        self.up_gold = 0  
        self.gold = 0  
        self.up_tech = 0  
        self.tech = 0  
        self.up_foot_att = 0  
        self.up_foot_def = 0  
        self.up_arch_att = 0  
        self.up_arch_def = 0  
        self.up_cav_att = 0  
        self.up_cav_def = 0  
        self.foot = 0  
        self.arch = 0  
        self.cav = 0
```

Listing 23 - klasa Forward()

Klasa ta zawiera zmienne, które zbierają informacje o tym co zrobił gracz w tej turze - liczbę rekrutowanych jednostek, które ulepszenia gracza opracowywuje oraz stan złota i punktów technologii pod koniec tury. Zmienne te są wysyłane na serwer po kliknięciu w przycisk końca tury, a sama klasa jest tworzona na nowo.

Klasa Mgame() z pliku run.py łączy wszystkie klasy gry ze sobą. W konstruktorze klasy widocznym na listingu 24:

1. Inicjalizowana jest biblioteka pyGame;
2. Definiowana jest rozdzielcość ekranu oraz tworzony jest sam ekran;
3. Tworzone jest połączenie z serwerem;
4. Tworzone są klasy Forward(), Board(), Cost(), Stats() oraz GFX();

5. Tworzone są zmienne odpowiedzialne za przechowywanie aktualnej podpowiedzi;

```
class Mgame():  
    def __init__(self, ip):  
        pygame.init()  
        self.window_size = (1800,800)  
        self.channel = grpc.insecure_channel(ip)  
        self.stub = game_service_pb2_grpc.TTTStub(self.channel)  
        self.screen = pygame.display.set_mode(self.window_size)  
        self.forward = Forward()  
        self.board = Board()  
        self.costs = Cost()  
        self.stats = Stats()  
        self.gfx = GFX()  
        self.tooltip_1 = ""  
        self.tooltip_2 = ""  
        self.tooltip_cost1 = 0  
        self.tooltip_cost2 = 0
```

*Listing 24 - konstruktor klasy Mgame()*

Po stworzeniu klasy uruchamiana jest metoda title. Jest ona odpowiedzialna za:

1. Namalowanie na ekranie grafiki ekranu startowego;
2. Tworzenie przycisków oraz dodanie do nich tekstu;
3. Obsługiwanie zdarzeń związanych z najechaniem i kliknięciem w przycisk:
  - a. Gdy użytkownik najedzie na przycisk jego środek ma zmienić kolor;
  - b. Gdy użytkownik kliknie w przycisk wyjścia program się wyłącza;
  - c. Gdy użytkownik kliknie w przycisk nowej gry metoda zwraca odpowiednią wartość - program wraca do funkcji main;

Gdy użytkownik podejmuje decyzje, że chce rozpocząć grę sprawdzane jest jaki tryb gry został włączony. Gdy tryb normalny to uruchamiana jest metoda matchmaking. Metoda ta zaciemnia ekran i wysyła na serwer komunikat, że nowy gra ma chęć na rozgrywkę. Mogą się wtedy wydarzyć dwa scenariusze:

1. W przypadku gdy na serwerze ktoś już oczekiwali na przeciwnika to zwróci on id gry, numer gracza oraz informacje że metoda może się zakończyć;
2. Jeżeli nikt nie oczekuje na przeciwnika to serwer zwróci id gry, numer gracza oraz informację że należy chwilę poczekać. Wtedy metoda co 2 sekundy wysyła na serwer zapytanie czy już się ktoś znalazł. Jeżeli nie to zapytanie

zostanie ponowione za 2 sekundy aż do momentu odpowiedzi pozytywnej od serwera. Wtedy metoda się zakończy.

Gdy metoda się kończy zwraca funkcji main id gry i numer gracza. A następnie te informacje są przekazywane do metody game. Gdy zostaw włączony tryb testowy to pomijana jest metoda matchmaking i od razu uruchamiana jest metoda game z parametrami id\_gry = "test" i numer\_gracza = 1.

Metoda game ma proste, ale bardzo ważne zadania:

1. Zmienia rozmiar i tworzy okno gry;
2. Zapisuje id gry i numer gracza w swojej klasie;
3. Rysuje po raz pierwszy panel lewy i prawy (definicje paneli są dostępne w rozdziale 3.4) za pomocą metod update\_left i update\_right;
4. W pętli pobiera aktualną pozycję myszki i przekazuje ją metodzie event\_handler, aż do momentu zwycięstwa bądź porażki

Gdy gra się zakończy wtedy metoda zwróci funkcji main komunikat, aby uruchomiła metodę title.

Update\_right i update\_left mają takie same zadanie - przerysowanie prawej/lewej części ekranu przy pomocy assetów i koordynat dostępnych w klasie GFX.

Rysowane są takie elementy jak([R] - gdy rysuje je metoda update\_right, [L] - gdy update\_left):

- tło ekranu [L] i [R];
- ikony zawieszenia broni/wojny oraz numer tury [L];
- ramki pól jednostek [L];
- jednostki na polach [L];
- banery [R];
- liczba złota i punktów technologii [R];
- ikony ulepszeń / rekrutacji jednostek / końca tury [R];
- podpowiedzi [R].

Każde przerysowanie musi wydarzyć się gdy któryś element interfejsu zostanie zmieniony, ale im częściej ono występuje tym więcej zasobów komputera gra zużywa dlatego należało to jakoś ograniczyć. Aby za każdym razem nie aktualizować całego ekranu został on podzielony na dwie części. Każdy element częściowej (zajmującej 70% ekranu) jest aktualizowany tylko na początku nowej tury. Każdy element częściowej musi być aktualizowany po kliknięciu przez

użytkownika w jakąś ikonę z tego panelu. Takie podzielenie ekranów sprawiło wzrost w wydajności gry.

Jak nazwa wskazuje metoda `event_handler` zajmuje się obsługą wszystkich zdarzeń wywołanych przez gracza. Aktualnie zdarzenie jest tylko jedno - kliknięcie w przycisk myszy. Po rozpoczęciu metoda sprawdza czy użytkownik kliknął przycisk myszy. Jeżeli tak się nie stało to metoda się kończy. Natomiast gdy użytkownik kliknął to wtedy sprawdzane są koordynaty myszy. Jeżeli są one w obrębie którejś ikony to sprawdzane jest który przycisk myszy został kliknięty:

- W przypadku środkowego przycisku do zmiennych odpowiedzialnych za przechowywanie podpowiedzi zostaną zapisane tooltipy przypisane do aktualnie klikniętej ikony;
- W przypadku lewego przycisku sprawdzane jest czy gracz ma wystarczająco punktów technologii lub/i złota aby przeprowadzona została akcja powiązana z ikoną. W przypadku ulepszeń sprawdzane dodatkowo jest czy nie zostało ono już wcześniej zakolejkowane. Jeżeli nie to odejmowane są z klasy `Board()` odpowiednie zasoby oraz w klasie `Forward()` zapisuje się informacje że gracz dokonuje konkretnego ulepszenia. W przypadku rekrutacji wojska w klasie `Forward()` zwiększa się o 1 zmienną odpowiedzialną za dany typ jednostki;
- Prawy przycisk myszy jest anulowaniem danej akcji. W przypadku ulepszeń sprawdzane jest czy w klasie `Forward()` użytkownik opracowuje dane ulepszenie. Jeżeli tak to jest ono anulowane, a graczowi zwracane są zasoby. W przypadku rekrutacji wojska sprawdzane jest czy w klasie `Forward()` zmienna odpowiedzialna za dany typ jednostki jest większa od 0. Jeśli tak to zmniejsza się jej stan o 1 i zwracane są graczowi zasoby.

Następnie wywoływana jest metoda `update_right`. W przypadku gdy koordynaty myszki nie są w obrębie jakiejkolwiek ikony to do zmiennych odpowiedzialnych za podpowiedź przypisywany jest pusty tekst. Wyjątek stanowi ikona końca tury. W jej przypadku sprawdzane jest tylko czy użytkownik kliknął w środkowy przycisk myszy. Jeżeli tak to wyświetlany jest odpowiedni tooltip. Jeżeli tył innego przycisku to niezależnie od tego który to był przycisk rozpoczęta jest akcja zakończenia tury która przebiega następująco:

1. Następuje przyciemnienie ekranu;

2. Do klasy Forward() zostaje zapisana liczba złota i punktów technologii z klasy Board();
3. Wywołana zostaje metoda end\_turn;
4. Co 2 sekundy wywoływana jest metoda ready\_check, aż do momentu gdy zwróci ona 1;
5. Wywoływana jest metoda get\_new\_turn;
6. Klasa Forward() jest tworzona od nowa;
7. Gdy któryś gracz zwyciężył zakończ rozgrywkę;
8. Wywołaj metody update\_right i update\_left.

Po zakończeniu akcji końca tury rozpoczyna się nowa tura.

Metoda end\_turn zamienia klasę Forward() na wiadomość Forward\_to\_serwer z pliku .proto i wysyła ją do serwera.

Metoda ready\_check wysyła do serwera wiadomość Next\_Turn\_Status w celu sprawdzenia czy drugi gracz wykonał już swoją turę i czy serwer obliczył nową.

Zwraca ona 0 w przypadku gdy gracz musi jeszcze poczekać na drugiego lub 1 gdy można pobrać stan nowej tury.

Metoda get\_new\_turn pobiera z serwera stan nowej tury przy pomocy wiadomości Next\_Turn oraz zapisuje uzyskane dane w klasie Board(). Metoda zwraca informacje o tym czy któryś z graczy wygrał(1 w przypadku gdy gracz wygrał, 2 gdy przegrał lub 0 gdy rozgrywka nie została rozstrzygnięta).

## Zasady Gry

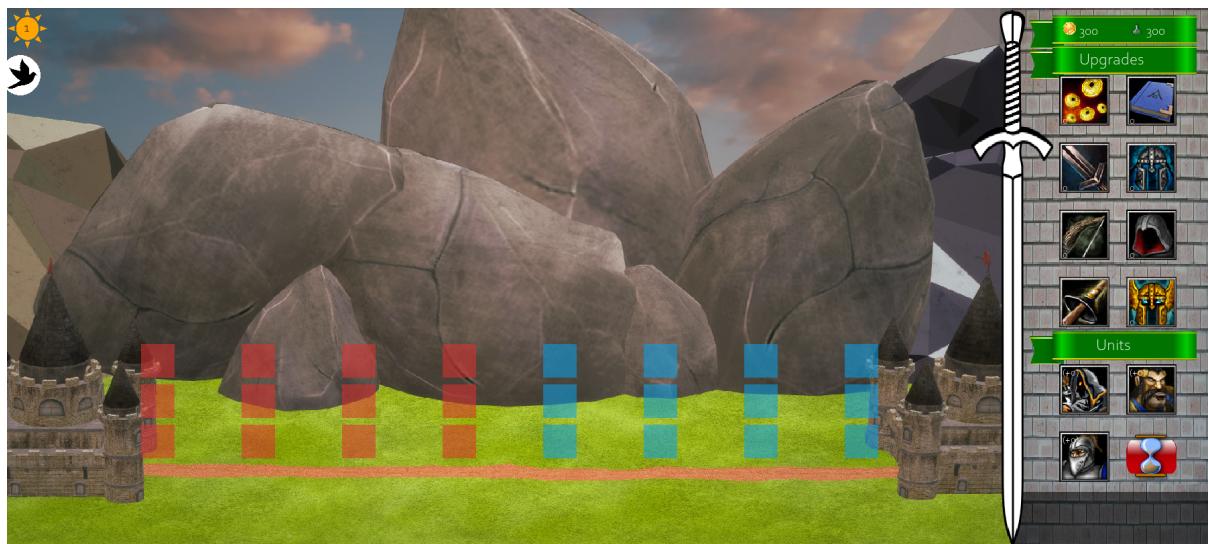


Po uruchomieniu gry pierwszym widokiem, który zobaczy gracz będzie ekran rozpoczęcia gry - rysunek 12. W zależności od trybu w którym została uruchomiona gra wydarzyć się mogą 2 rzeczy:

1. W normalnym trybie ekran zmieni wielkość i stanie się czarny co zwiastować będzie rozpoczęcie procesu szukania przeciwnika. Po znalezieniu go gra wczyta główny ekran;
2. W trybie testowym pominięte zostanie szukanie przeciwnika. Gra tylko wczyta główny ekran.

Rysunek 12 - Ekran Rozpoczęcia Gry

Główny ekran gry - przedstawiony na rysunku 13 - składa się z dwóch części:



Rysunek 13 - Główny Ekran Gry

1. Panelu lewego - po lewo od białego miecza - jego zadaniem jest wyświetlanie najważniejszych dla gracza informacji. Do nich należą:
  - rozmieszczenie swoich i wrogich wojsk - Rysunek 14;



Rysunek 14 - pola bitwy

- między którymi polami będzie rozgrywała się aktualna bitwa - Rysunek 15;
- numer tury - Rysunek 16;
- czy aktualnie trwa zawieszenie broni - Rysunek 17.



Rysunek 17 - ikony wojny/rozejmu



Rysunek 16 - ikona numeru tury



Rysunek 15 - pola sporne



Rysunek 18 - stan złota/punktów technologii



Rysunek 19 - zakładka ulepszeń

2. Panelu prawego - po prawo od białego miecza - to na nim gracz wykonuje wszystkie swoje akcje. Na samej jego górze - Rysunek 18 - wyświetlany jest aktualny stan złota oraz punktów technologii. Poniżej - Rysunek 19 - znajduje się zakładka ulepszeń dla jednostek i stanu ekonomii. W pierwszej kolumnie tymi ulepszeniami są:

- Ulepszenie przychodu złota;
- Ulepszenie obrażeń piechoty;
- Ulepszenie obrażeń łuczników;
- Ulepszenie obrażeń kawalerii.

W drugiej kolumnie widnieje:

- Ulepszenie przychodu punktów technologii
- Ulepszenie pancerza piechoty;
- Ulepszenie pancerza łuczników;

- Ulepszenie pancerza kawalerii.

Pod panelem ulepszeń znajduje się menu rekrutacji jednostek oraz przycisk końca tury - Rysunek 20. Można tutaj zaobserwować ikonę rekrutacji łuczników (lewy górny róg), ikonę rekrutacji kawalerii (prawy górny róg), ikonę rekrutacji piechoty (lewy dolny róg) oraz ikonę końca tury (prawy dolny róg). Na samym dole wyświetlany jest tooltip (małe okienko z informacjami powiązane z danym elementem interfejsu użytkownika) - Rysunek 21 - wyświetlany jest po najechaniu myszką na dowolną ikonę w panelu ulepszeń lub rekrutacji i naciśnięciu środkowego przycisku myszy.



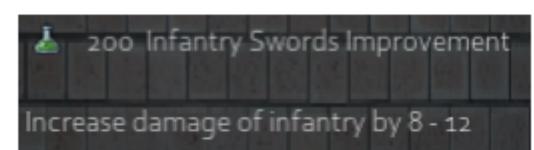
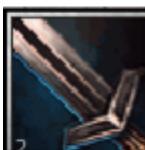
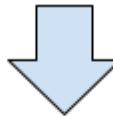
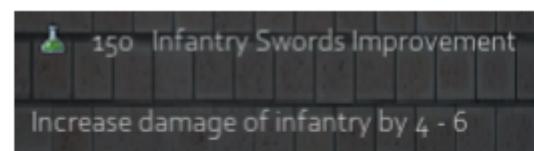
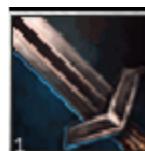
Rysunek 20 - panel rekrutacji



Rysunek 21 - tooltip

## Cel gry

Celem gry jest zajęcie wszystkich ośmiu pól (z rysunku 14 w panelu lewym). Gracz przegrywa gdy to jego wróg zajmie wszystkie pola. Aby sprostać temu wyzwaniu musi wykorzystać narzędzia udostępnione w panelu prawym. W każdej turze gracz dostaje przychód w postaci punktów technologii oraz złota. Można go zwiększyć wynajdując odpowiednie ulepszenia. Punkty technologii wykorzystywane są podczas wynajdywania ulepszeń. Złoto wydawane jest podczas rekrutacji jednostek. Każde ulepszenie ma swój poziom - zwiększając go



Rysunek 22 - różne poziomy i koszt ulepszenia

zwiększają się statystyki jednostek ale także ich koszt jak i koszt samego ulepszenia  
- Rysunek 22. Aby efekt ulepszeń zaczął działać gracz musi rozpocząć nową turę.

Aby zdobyć sporne pole gracz musi zrekrutować wojsko. Gdy gracz doda do kolejki rekrutacji wojsko zostanie ono wystawione przy jego zamku po rozpoczęciu nowej tury, a każde wojsko już wystawione zostanie przesunięte o 1 pole w lewo (aż do pola spornego) - Rysunek 23.



Rysunek 23 - przesunięcie jednostek między turami

Wojsko jest podzielone na 3 klasy: Piechotę, Łuczników i Kawalerię. Każda klasa jest wystawiana w oddzielnym polu w danej kolumnie - jak jest to widoczne na Rysunku 12X - piechota na samej górze, łucznicy na środku oraz kawaleria na dole. Aby zająć pole sporne należy pokonać piechotę wroga (i tylko piechotę - nie trzeba pokonywać łuczników ani kawalerii). Każda jednostka może atakować jednostki tego samego poziomu lub niższego. Poziomy jednostek wyglądają następująco: Piechota - 1, Łucznicy - 2, Kawaleria - 3. Po pokonaniu piechoty któregoś z graczy linia sporna zostanie przesunięta w stronę pokonanego gracza, a wszystkie pozostałe przy życiu jednostki łuczników i kawalerii przegranej wycofają się na nowe pole sporne. Zostanie wtedy podpisanie 5 dniowe (pięcio turowe) zawieszenie broni. Oznacza to jednostki między polami spornymi nie będą walczyć przez 5 tur. Da to

czas obu graczom na przysłanie dodatkowych jednostek piechoty. Czas zawieszenia broni nie jest nigdzie graczom komunikowany oznacza to że gracze muszą zapamiętać kiedy zawieszenie zostało podpisane jeśli nie chcą zostać zaskoczeni. Gra się zakończy gdy któryś z graczy zajmie wszystkie pola. Wtedy gra powróci do ekranu startowego.

#### Uwaga!

Jeżeli gra zostanie uruchomiona w trybie testowym to gracz nie będzie miał przeciwnika, a przycisk końca tury sprawi że gra w nieskończoność będzie oczekiwana to turę przeciwnika. Aby tego uniknąć należy wykorzystać plik `client_test.py` dostępny w folderze serwera.

#### Uwaga 2!

Po kliknięciu w przycisk nowa gra na ekranie startowym - Rysunek X - okno gry się rozszerzy w prawą stronę. Wszystko co nie zmieści się na ekranie monitora nie zostanie poprawnie wyrenderowane. Dlatego zalecane jest aby ekran startowy trzymać blisko lewej górnej krawędzi ekranu.