Deathmatch

Krzysztof Nasuta 193328 Aleksander Iwicki 193354 Filip Dawidowski 193433

1. Wstęp

Celem projektu było stworzenie gry deathmatch, w której gracze mogą walczyć ze sobą na arenie. Do stworzenia gry użyto języka C++ oraz biblioteki Boost. Gra działa w trybie tekstowym, sterowanie odbywa się za pomocą klawiatury. Gracze mogą poruszać się po arenie, strzelać do siebie oraz zbierać ulepszenia (np szybsze strzelanie). Grę wygrywa gracz, który zdobędzie wcześniej określoną liczbę punktów, które zdobywa się poprzez eliminację przeciwników.

1.1. Działanie z punktu widzenia użytkownika

Na początku gracz wpisuje IP serwera, na którym chce grać. Następnie gracz wybiera nick, którym będzie się posługiwał w grze. Po wybraniu nicku gracz dołącza do lobby gdzie widzi wszystkich innych graczy i może zaznaczyć gotowość. Po zaznaczeniu gotowości przez wszystkich użytkowników rozpoczyna się rozgrywka. Gracz może poruszać się po arenie za pomocą klawiszy W, A, S, D i strzelać za pomocą strzałek. Po wejściu na pole z ulepszeniem gracz je automatycznie podnosi. Za każdą eliminację przeciwnika gracz zdobywa punkt. Grę wygrywa gracz, który zdobędzie wcześniej określoną liczbę punktów.

1.2. Działanie z punktu widzenia serwera

Po połączeniu się klientów serwer rozpoczyna grę. Oczekuje na akcje od klientów, które od razu propaguje do klientów, celem zmniejszenia opóźnień. Serwer symuluje stan gry, przeprowadza symulację ticku oraz synchronizuje stan gry klientów. Kolejność symulacji jest ustalona i nie pozwala na sytuacje wyścigu. Po zakończeniu gry serwer wysyła informację o zakończeniu gry do klientów, którzy mogą zagrać kolejny mecz lub opuścić grę.

2. Przypuszczalne problemy

2.1. Co się stanie jak gracz wejdzie w nieśmiertelność i zostanie trafiony w tym samym czasie

Wszystkie wydarzenia w grze będą przetwarzane w kolejce priorytetowej i wszystkie ulepszenia wykonają się przed sprawdzaniem trafień, analogicznie jeśli w tym samym momencie gracz się ruszy, i zostanie trafiony to najpierw przetworzone zostaną wszystkie ruchy graczy, a dopiero potem sprawdzanie trafień.

2.2. Niska responsywność gry

Ze względu na niską częstotliwość odświeżania konsoli w trybie tekstowym, gra może wydawać się nie responsywna i często może dochodzić do sytuacji gdzie dwóch graczy w jednym momencie robią ruch na to samo pole (wtedy się zderzają i obydwoje zatrzymują), ponieważ między pojedynczymi aktualizacjami może minąć dużo czasu. Aby temu zaradzić sama gra będzie się odświeżać 60 razy na sekundę niezależnie od prędkości odświeżania konsoli, dzięki czemu dużo częściej o wyniku jakiejś akcji będzie decydować zręczność gracza, a nie to czy akurat w tym momencie konsola się odświeżyła.

2.3. Co jeśli pakiety UDP nie przyjdą w kolejności

Samo zdarzenie jest mało prawdopodobne, ponieważ pakiety są wysyłane tylko w momencie gdy gracz kliknie lub zwolni przycisk, co musiałoby się stać szybciej niż opóźnienie w dostarczeniu pakietu. W przypadku gdyby jednak tak się stało, to co określony czas, najprawdopodobniej co każde odświeżenie konsoli serwer będzie wysyłał stan gry, co pozwoli na synchronizację stanu gry między klientami oraz serwerem.

2.4. Co gdyby gracz zmienił kod gry i zaczął wysyłać fałszywe pakiety

Gracz wysyła tylko informacje o akcjach które wykonuje (wysyła np “zaczynam iść do przodu”, a nie “moja pozycja to X Y”), a serwer sam symuluje stan gry, który potem rozsyła innym klientom. Dzięki takiemu rozwiązaniu oszukiwanie nie będzie możliwe.

3. Struktura komunikatów

Każdy komunikat zawiera dodatkowo umieszczone na początku 1-bajtowe pole *packet_id* oznaczające jego rodzaj.

Dołączenie do gry (Klient -> Serwer)		
Nazwa pola	**Typ danych**	**Opis**
nick_length	int32	Długość nicku gracza (maksymalnie: 20)
nick	char[]	Nick gracza o długości nick_length

Przesłanie informacji o mapie (Serwer -> Klient)		
Nazwa pola	**Typ danych**	**Opis**
player_id	int32	id, jakie zostało nadane temu graczowi
width	int32	Szerokość mapy
height	int32	Wysokość mapy
tiles	MapTile[]	Tablica kafelków mapy o rozmiarach width*height, każdy kafelek ma rozmiar 4 bajtów

Informacja o gotowości gracza (Klient -> Serwer)
Przesyłany jest jedynie odpowiedni *packet_id*

Informacja o rozpoczęciu gry (Serwer -> Klient)
Przesyłany jest jedynie odpowiedni *packet_id*

Akcja od gracza [UDP] (Klient -> Serwer)		
Nazwa pola	**Typ danych**	**Opis**
action	int32(enum)	Jedna z wartości: 0 - poruszanie się w górę, 1 - poruszanie się w dół, 2 - poruszanie się w lewo, 3 - poruszanie się w prawo, 4 - wystrzelenie pocisku w górę, 5 - wystrzelenie pocisku w dół, 6 - wystrzelenie pocisku w lewo, 7 - wystrzelenie pocisku w prawo, 8 - zatrzymanie ruchu gracza

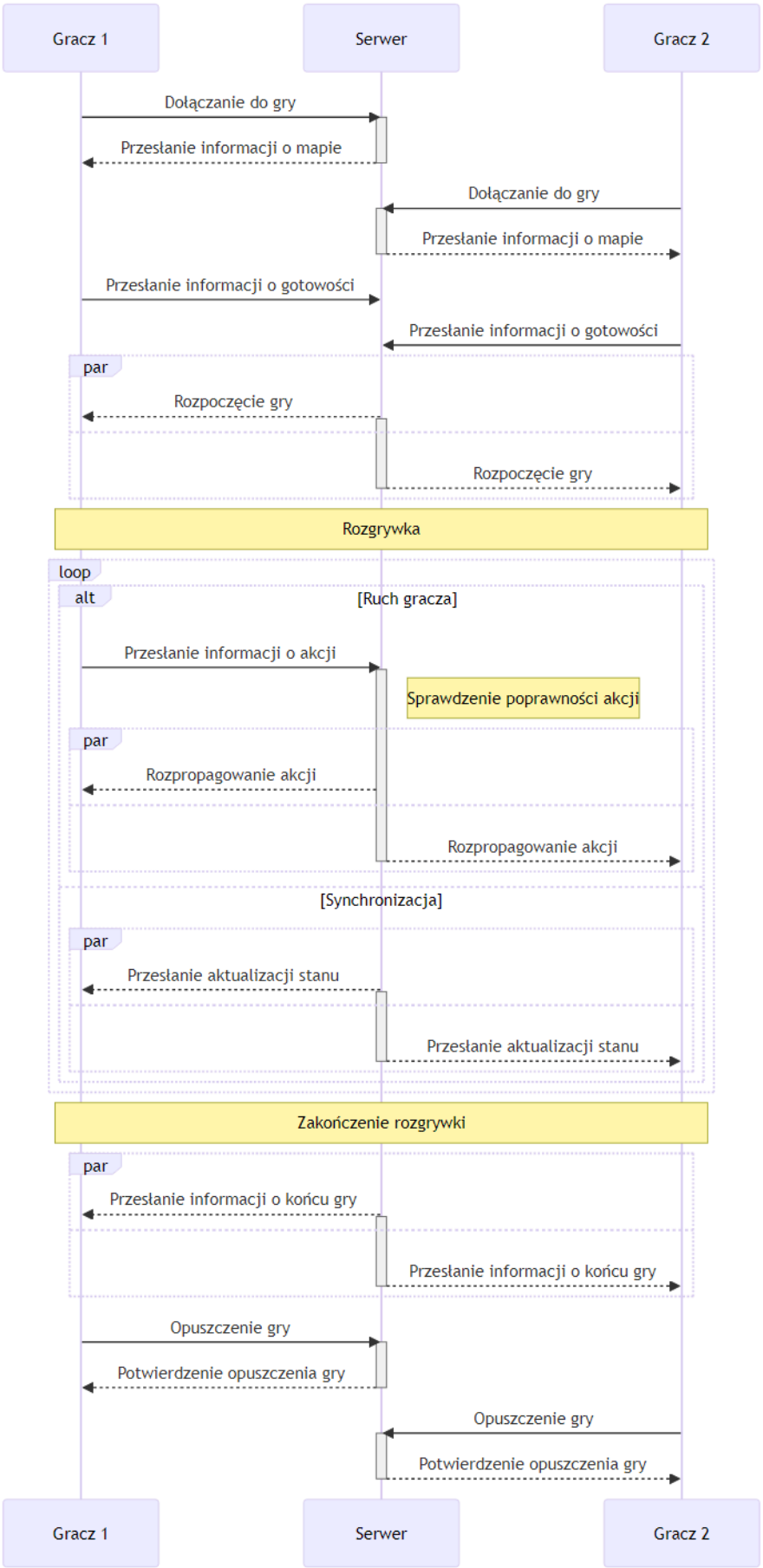
Propagacja akcji [UDP] (Serwer -> Klient)		
Nazwa pola	**Typ danych**	**Opis**
action	int32(enum)	Jedna z wartości: 0 - poruszanie się w górę, 1 - poruszanie się w dół, 2 - poruszanie się w lewo, 3 - poruszanie się w prawo, 4 - wystrzelenie pocisku w górę, 5 - wystrzelenie pocisku w dół, 6 - wystrzelenie pocisku w lewo, 7 - wystrzelenie pocisku w prawo, 8 - zatrzymanie ruchu gracza
player_id	int32	id gracza, którego dotyczy akcja

Informacja o końcu gry (Serwer -> Klient)		
Nazwa pola	**Typ danych**	**Opis**
scores_len	int32	Długość tablicy wyników
scores	PlayerScore[]	Tablica struktur PlayerScore. Pola struktury: *player_id* - id gracza (int32) *score* - wynik gracza (int32) Rozmiar: 8 bajtów

Informacja o opuszczeniu gry (Klient -> Serwer)
Przesyłany jest jedynie odpowiedni *packet_id*

Synchronizacja stanu gry (Serwer -> Klient)		
Nazwa pola	**Typ danych**	**Opis**
players_num	int32	Ilość graczy w grze
players	Player[]	Tablica struktur Player. Pola struktury: *player_id* - id gracza (int32) *x, y* - koordynaty gracza (2*int32) *score* - wynik gracza (int32) Rozmiar: 16 bajtów
bullets_num	int32	Ilość aktualnie wystrzelonych pocisków
bullets	Bullet[]	Tablica struktur Bullet. Pola struktury: *owner_id* - id gracza, który wystrzelił pocisk (int32) *x, y* - koordynaty pocisku (2*int32) Rozmiar: 12 bajtów
width	int32	Szerokość mapy
height	int32	Wysokość mapy
tiles	MapTile[]	Tablica kafelków mapy o rozmiarach width*height, każdy kafelek ma rozmiar 4 bajtów

4. Diagram sekwencji



4.1. Schemat działania

Przed rozpoczęciem gry serwer tworzy nową mapę. Klienci łączą się z serwerem oraz przesyłają informację o swoim nicku w grze. Serwer potwierdza dołączenie do poczekalni, jeśli są w niej wolne miejsca, oraz przesyła użytkownikowi informację o mapie, na której będzie rozgrywany mecz. Klienci mogą zgłaszać się do gry aż do jej rozpoczęcia. Serwer oczekuje na otrzymanie od każdego z zarejestrowanych graczy potwierdzenia gotowości. Kiedy otrzyma je od każdego z klientów, rozpoczyna się rozgrywka.

W trakcie gry serwer oczekuje na akcje od klientów. Dozwolonymi akcjami jest zmiana stanu ruch (rozpoczęcie poruszania w jednym z 4 kierunków bądź zatrzymanie) lub strzał (w wybranym z 4 kierunków). Po odebraniu akcji serwer waliduje jej poprawność i umieszcza w kolejce rozkazów. Serwer propaguje akcję do pozostałych klientów. Pozwala to na szybszą aktualizację stanu gry u innych graczy i usprawnienie rozgrywki. Należy pamiętać, że stan gry jest symulowany na serwerze. Rozpropagowanie akcji ma na celu obniżenie opóźnień w grze, ale nie wpływa na rozgrywkę w inny sposób.

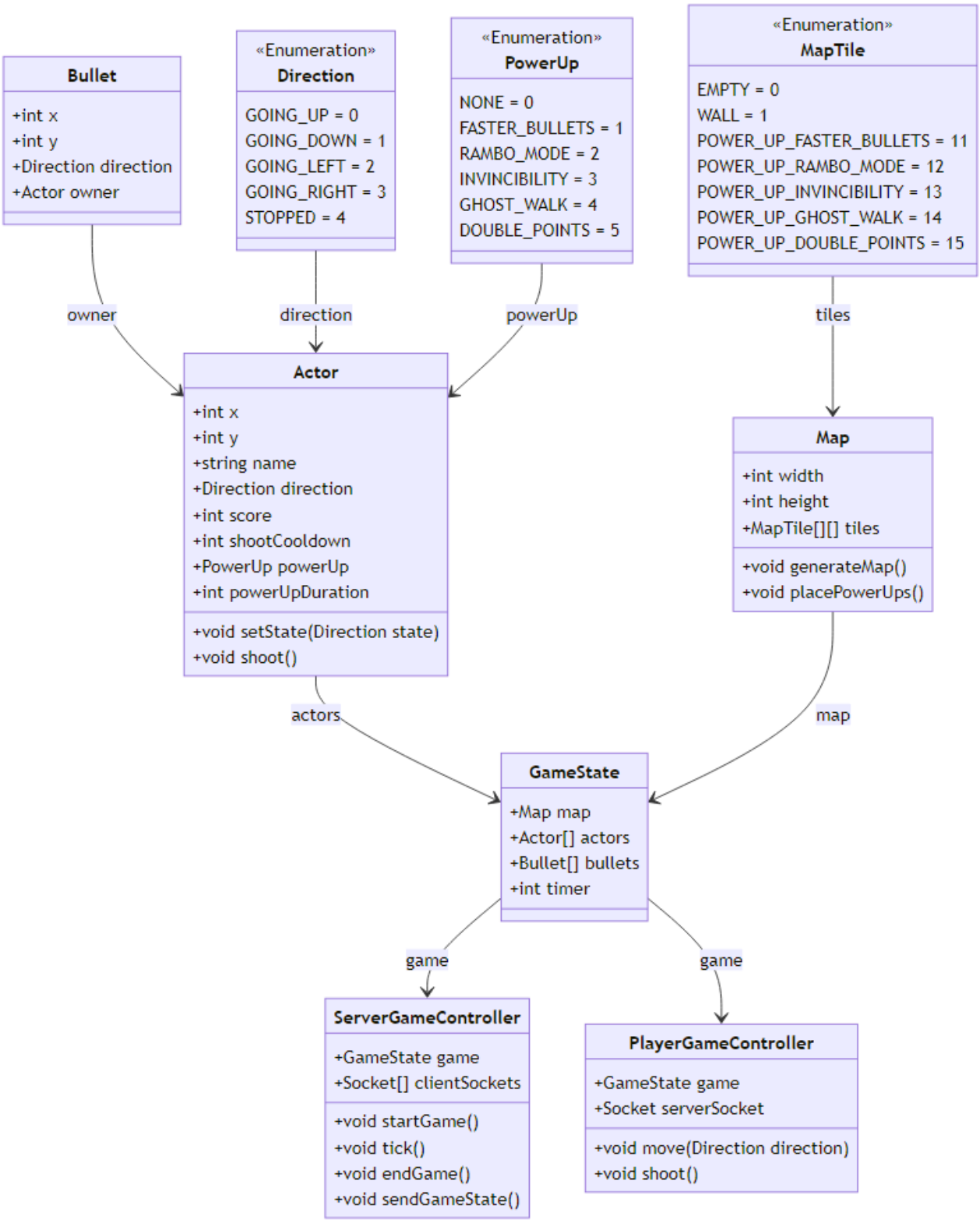
Co każdy tick serwera (czyli co określony czas) serwer wykonuje wszystkie akcje z kolejki rozkazów. Kolejka ta jest implementacją priorytetową, co pozwala na zachowanie kolejności akcji (np wszystkie akcje ruchu w tym samym ticku wykonają się przed wszystkimi akcjami strzelania). Po wykonaniu wszystkich akcji serwer przesyła zaktualizowany stan gry do wszystkich klientów. Pozwala to na synchronizację stanu gry między klientami oraz serwerem.

Następnie przeprowadzana jest symulacja ticku.

- Na początku przesuwamy każdego z graczy w kierunku, w którym się porusza. Kolejność przesuwania graczy jest stała.
- Sprawdzane są kolizje z innymi graczami. Jeśli do niej dojdzie, stan wszystkich graczy w niej uczestniczących zamieniany jest na brak ruchu.
- Następnie sprawdzana jest kolizja ze ścianami mapy. W przypadku próby wejścia w ścianę, gracz jest zatrzymywany. Jeśli gracz ma ulepszenie, które pozwala na przechodzenie przez ściany, to nie jest zatrzymywany. Jeśli ulepszenie to się skończyło, a gracz jest "w ścianie", przesuwamy go na najbliższe wolne pole według ustalonego algorytmu.
- Sprawdzamy kolizje z ulepszeniami. Jeśli gracz wejdzie na pole z ulepszeniem, gracz podnosi je, a ulepszenie znika z mapy. Jeśli gracz już ma inne ulepszenie, to zostaje ono zastąpione nowym.
- Następnie przesuwane są pociski, które sprawdzają kolizje z graczami oraz ścianami. W przypadku kolizji z graczem, gracz ginie, a strzelec zdobywa punkt. W przypadku kolizji z ścianą pocisk znika.
- Jeśli gracz zginął w tym ticku, to jest on przesuwany na ustalone pole odrodzenia, jeśli jest zajęte to wybierane jest najbliższe puste pole według ustalonego algorytmu.

Po upłynięciu określonego czasu gry mecz kończy się. Serwer wysyła informację o zakończeniu gry do wszystkich klientów. Klienci otrzymują informację o zwycięzcy oraz wynikach meczu. Klienci mogą ponownie potwierdzić gotowość i zagrać kolejny mecz, lub opuścić poczekalnię i rozłączyć się z serwerem.

5. Diagram klas



5.1. Opis klas

- Direction - enum reprezentujący kierunek ruchu. Dozwolonymi kierunkami są: góra, dół, lewo, prawo oraz brak ruchu.
- PowerUp - enum reprezentujący ulepszenie. Dozwolonymi ulepszeniami są: szybsze strzelanie, strzelanie na wszystkie strony, nieśmiertelność, przechodzenie przez ściany, podwójne punkty za zabicie przeciwnika.
- MapTile - enum reprezentujący kafelek mapy. Dozwolonymi kafelkami są: ściana, wolne pole, pole z ulepszeniem.
- Bullet - klasa reprezentująca pocisk. Przechowuje informacje o kierunku, w którym się porusza, oraz o graczu, który go wystrzelił.
- Actor - klasa reprezentująca gracza. Przechowuje informacje o pozycji, nicku, kierunku ruchu, ulepszeniach, punktach oraz czasie odnowienia strzału.
- Map - klasa reprezentująca mapę. Przechowuje informacje o szerokości, długości, kafelkach oraz ulepszeniach na mapie.
- GameState - klasa reprezentująca stan gry. Przechowuje informacje o mapie, graczach, pociskach oraz czasie gry.
- ServerGameController - klasa odpowiedzialna za zarządzanie grą na serwerze. Przechowuje informacje o stanie gry oraz podłączonych klientach. Zarządza grą, odbiera akcje od klientów, przetwarza je oraz przesyła stan gry do klientów.
- PlayerGameController - klasa odpowiedzialna za zarządzanie grą na kliencie. Przechowuje informacje o stanie gry oraz połączeniu z serwerem. Zarządza grą, wysyła akcje do serwera, odbiera stan gry oraz zleca rysowanie stanu gry.