13 Maja 2025 r.

Przetwarzanie rozproszone

„Historia mądrością – przyszłość wyzwaniem"

Implementacja gry sieciowej z wykorzystaniem gniazd systemu Linux

Sprawozdanie do projektu – część 1, 2 i 3

Bartosz Skorowski 197757 | Kamil Raubo 198328 | Jakub Bot 197839

POLITECHNIKA GDAŃSKA

13 Maja 2025 r.

Sprawozdanie – część pierwsza

Przetwarzanie rozproszone

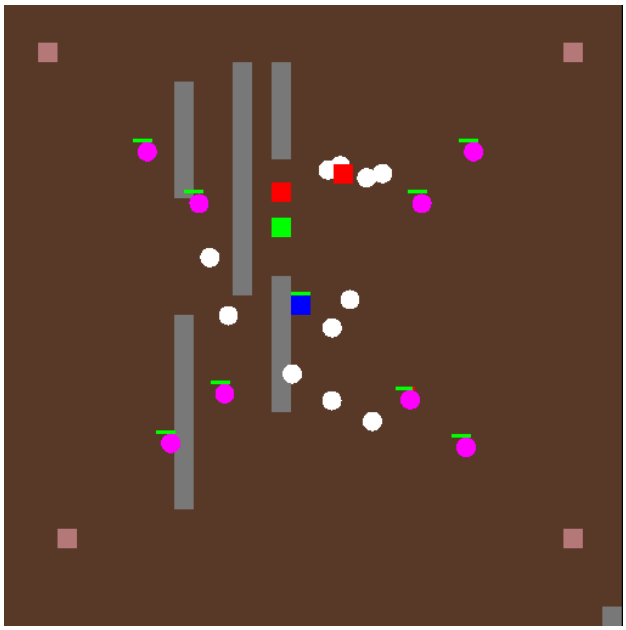
Bartosz Skorowski 197757 | Kamil Raubo 198328 | Jakub Bot 197839

Spis treści

Opis gry 4
Zasady gry 5
Przebieg rozgrywki 5
Sterowanie 6
Architektura i aspekty sieciowe 6
Metoda komunikacji 7
Role serwera oraz klienta 7
Sytuacje sporne 8
Sekcje krytyczne 8

Opis gry

Gra jest typu “Tower Defense”. Gracze wspólnie bronią określonego punktu, będącego bazą, przed atakami przeciwników (AI). W celu pokonania przeciwników gracze mogą strzelać kulami ognia, które zadają obrażenia przeciwnikom.



Rysunek 1 - Przykładowy wygląd gry.

Rysunek 1 przedstawia przykładowy wygląd gry. Czerwone kwadraty – gracze, różowe koła – przeciwnicy z paskami życia, niebieski kwadrat – baza graczy, białe koła – kule ognia, jasnobrązowe kwadraty – spawnery przeciwników, szare pola – ściany (elementy dekoracyjne), zielony kwadrat określa punkt startowy graczy.

Zasady gry

- Gra przewidziana jest dla 2-4 graczy.
- Gra trwa do momentu porażki lub zwycięstwa graczy.
- Gracze rozpoczynają grę w wyznaczonych punktach.
- Gracze mogą atakować przeciwników za pomocą strzałów ognistymi kulami.
- Przeciwnicy kierują się w stronę bazy graczy, ignorując ich samych.

Przebieg rozgrywki

Gracze, po pojawieniu się na mapie zaczynają grę pokonując przeciwników, tworzących się w określonych punktach na mapie co ustalony okres. Ilość przeciwników jest ustalona dla każdego poziomu, a gdy gracze pokonają każdego - zwyciężają. W Przypadku, gdy przeciwnicy dotrą do bazy graczy zadają jej obrażenia zmniejszając tym samym jej ilość punktów życia. Gdy punkty życia bazy spadną poniżej 0, gracze przegrywają. W celu pokonania przeciwników gracze muszą jak najszybciej zacząć atakować przeciwników, tak by nie dotarli do bazy.

Sterowanie

- Poruszanie odbywa się za pomocą klawiszy **W**, **A**, **S** oraz **D.**
- Kula ognia wystrzeliwana jest po naciśnięciu **lewego przycisku myszy**.

Architektura i aspekty sieciowe

Wybrana została architektura Klient-Serwer. Architektura ta pozwala odciążyć graczy i przerzucić więcej obowiązków na serwer gry. Model ten ułatwia również komunikację z innymi graczami, ponieważ klient komunikuje się tylko z serwerem i nie musi zarządzać połączeniami ze wszystkimi graczami (jak ma to miejsce w przypadku architektury peer-to-peer), co znacząco zmniejsza liczbę krawędzi w grafie połączeń.

Serwer jest autorytarny i to on zarządza rzeczywistym światem gry, walidując akcje i ruchy graczy oraz symulując ruchy przeciwników i innych jednostek. Gracze wysyłają do serwera komunikaty przy wykonaniu akcji, jednocześnie symulując lokalny świat, tak by rozgrywka wydawała się płynna i responsywna. Serwer co określony czas wysyła dane do klientów, synchronizując ich.

Metoda komunikacji

Cała komunikacja opiera się o interfejs gniazd systemu Linux z użyciem protokołu TCP. Protokół ten został wybrany nad protokołem UDP między innymi ze względu na łatwość implementacji. UDP jest szybszy, lecz bardziej zawodny. W przypadku projektowanej gry prędkość oraz ilość przesyłanych danych nie stanowiła problemu.

Role serwera oraz klienta

Serwer:

- Jest autorytarny – to on rozstrzyga konflikty i zarządza grą.
- Przetwarza komunikaty od graczy i aktualizuje stan gry.
- Zajmuje się synchronizacją wszystkich graczy.
- Waliduje akcje graczy.

Klient:

- Wyświetla aktualny stan gry.
- Ma kontrolę na ruchami swojej postaci.
- Pomiędzy komunikatami od serwera lokalnie symuluje świat.

Sytuacje sporne

Sytuacje sporne rozwiązywane są przez serwer

- Dwóch graczy atakuje tego samego przeciwnika w tym samym czasie (ticku): Do każdego komunikatu klienta dołączany jest znacznik czasu akcji, co pomaga rozstrzygnąć, który gracz wykonał akcję. W przypadku gdy rozwiązanie to nie daje jednoznaczności można wylosować jednego z graczy, co nie pogorszy sytuacji żadnego z graczy, gdyż grają oni w jednej drużynie.
- Rozłączenia gracza z serwerem: W przypadku rozłączenia gracza serwer usuwa go z listy połączonych graczy oraz informuje o tym pozostałych graczy.
- Próby oszustwa: Serwer waliduje poprawność żądań od graczy, co prowadzi do znacznego utrudnienia prób oszustw przez graczy.

Sekcje krytyczne

W celu uniknięcia trudnych do wykrycia błędów sekcje krytyczne powinny być obsługiwane przez jeden wątek, by uniknąć tzw. konfliktów współbieżności (ang. race conditions). Przykładem takiej sytuacji jest zaatakowanie przeciwnika, zmniejszające jego punkty zdrowia lub pokonującego go (i usuwające z planszy).

Nie wszystkie akcje graczy mogą być przetwarzane współbieżnie, z tego powodu można osobno przetwarzać akcje bezpieczne (na przykład poruszenie się gracza) oraz potencjalnie niebezpieczne (na przykład aktualizację listy przeciwników po ataku przez gracza) i w przypadku sekcji krytycznych zastosować mechanizmy zapewniające, że tylko 1 wątek ma dostęp do modyfikacji danych.

Sprawozdanie – część druga

Przetwarzanie rozproszone

Bartosz Skorowski 197757 | Kamil Raubo 198328 | Jakub Bot 197839

Spis treści

Wprowadzenie 12
Ogólny schemat działania 12
Diagram sekwencyjny 13
Połączenie i inicjalizacja 15
Rozgrywka 15
Zakończenie gry 16
Diagram klas 16

Wprowadzenie

Projektowana gra sieciowa jest typem gry przypominającej “Tower Defense”. Głównym jej celem jest obrona bazy przez graczy. Baza ta atakowana jest przez przeciwników pojawiających się na mapie. Aby wygrać gracze muszą pokonać wszystkich wrogów zanim zniszczą oni bazę główną.

Ogólny schemat działania

Działanie gry można opisać w poniższych krokach.

1. Gracze łączą się z serwerem i dołączają do poczekalni.
2. Gracze wysyłają żądanie gotowości do serwera.
3. Gdy wszyscy gracze będą gotowi serwer rozpoczyna grę i wysyła każdemu klientowi informacje inicjalizujące (jego pozycję startową, i mapę).
4. Gracze wysyłają żądania przy wykonaniu akcji (ruch, strzał) do serwera i wykonują lokalną symulację rozgrywki.
5. Serwer wykonuje symulacje (obsługuje zadawanie obrażeń, tworzenie przeciwników) i na bieżąco wysyła zaktualizowany stan gry do graczy.

6. Kroki 6 oraz 7 wykonywane są do momentu zakończenia rozgrywki (porażki lub zwycięstwa).
7. Gdy rozgrywka dobiegnie końca gracze są o tym informowani oraz wracają do poczekalni, gdzie mogą zacząć kolejną rozgrywkę.

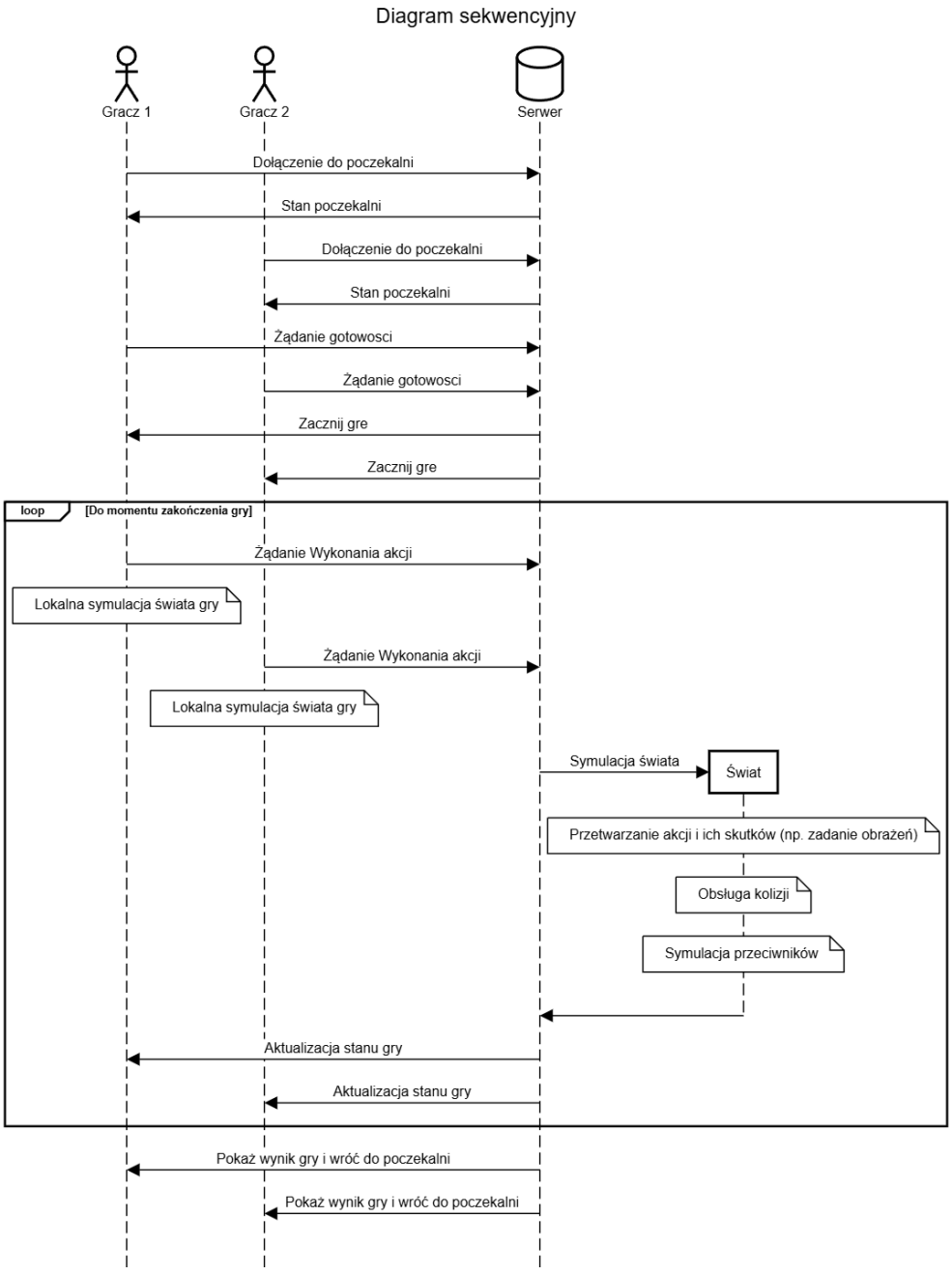
Diagram sekwencyjny

Poniższy diagram sekwencyjny przedstawia sytuacje w przypadku rozgrywki z 2 graczami (dla większej ilości graczy, część gracza powinna zostać analogicznie powielona).

Przedstawia on 3 główne fazy aplikacji:

1. Połączenie z serwerem i inicjalizacja
2. Rozgrywka - pętla gry
3. Zakończenie - działania po porażce lub zwycięstwie

Poszczególne fazy zostaną szczegółowo opisane w dalszej części.



Rysunek 2 - Diagram sekwencyjny rozgrywki.

Połączenie i inicjalizacja

Podczas tej sekcji klient łączy się z serwerem, po czym zostaje umieszczony w poczekalni wraz z innymi graczami. Gracze wysyłają serwerowi komunikat o gotowości do rozgrywki. Serwer informuje każdego członka poczekalni o pozostałych oraz o zmianie ich stanu gotowości. Gdy każdy z graczy wyrazi gotowość, gra rozpoczyna się - serwer wyznacza pozycje startowe każdego gracza, wybiera mapę i wysyła te informacje do klientów a następnie przechodzi do kolejnego stanu – rozgrywki.

Rozgrywka

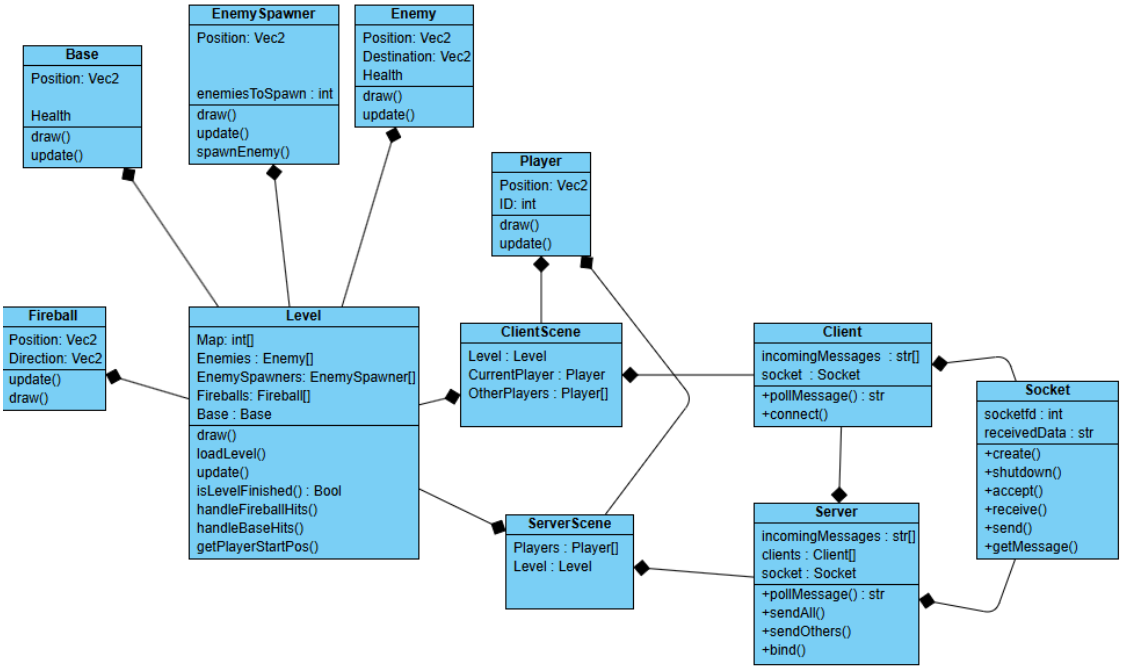
Faza ta trwa tak długo, aż gra nie zakończy się (porażką lub zwycięstwem). Klienci wysyłają do serwera informacje o wykonanych akcjach (np. Ruchu, strzale kulą ognia) i wykonują lokalne przewidywania gry (tak jakby ich akcja przeszła walidację na serwerze). W tym czasie, po otrzymaniu żądań serwer sprawdza je oraz odpowiednio aktualizuje świat gry, po czym przesyła aktualny stan klientom. Klienci po otrzymaniu danych z serwera odpowiednio aktualizują swój lokalny świat i w razie rozbieżności odpowiednio aktualizują dane.

Zakończenie gry

W momencie, gdy rozgrywka zakończy się, porażką lub zwycięstwem graczy, serwer informuje graczy o wyniku gry, a gracze wracają do poczekalni, gdzie mogą rozpocząć kolejną rundę gry, czy też zaprosić nowego gracza do poczekalni.

Diagram klas

Poniższy diagram klas ukazuje zaprojektowaną architekturę aplikacji.



Rysunek 3 - Diagram klas aplikacji.

Najważniejszym aspektem gry stworzonej w ramach projektu z przedmiotu jest wieloosobowa kooperacja graczy poprzez komunikacje z serwerem. W implementacji tego modelu wykorzystaliśmy sieciowe gniazda systemu Linux wykorzystujące protokół TCP.

Klasa **Socket** stanowi minimalną abstrakcję nad systemowym gniazdem, pozwalające na dwukierunkową komunikację z serwerem oraz na poprawne rozróżnianie wysłanych komunikatów.

Klasa **Serwer** zawiera informacje o podłączonych klientach oraz jest odpowiedzialna za wysyłanie i odbieranie informacji od klientów. Obsługuje ona również przypadki nagłych rozłączeń klientów od serwera.

Klasa **Client** to dodatkowa abstrakcja na klasie **Socket** ułatwiająca odczytywanie komunikatów od serwera.

Klasy **ClientScene oraz ServerScene** są to główne pętle gry klienta oraz serwera. Przetwarzane są w nich komunikaty od serwera/klientów, sprawdzane są warunki zakończenia gry oraz w przypadku klasy **ClientScene** znajduje się również obsługa klawiatury i myszy.

Klasa **Level** zarządza całym światem gry – aktualizuje wszystkie elementy oraz zajmuje się ich wyświetlaniem.

Klasa **Player** przedstawia obiekt sterowany za pomocą klawiatury i myszy przez graczy.

Klasa **Enemy** dsa przedstawia strukturę przeciwnika. Kieruje się on w stronę bazy, a gdy wystarczająco sie do niej zbliży zaatakuje ją zmniejszając jej poziom życia. Przeciwnicy posiadają życie, które zmniejsza się po kolizji przeciwnika z **Fireball.**

Klasa **EnemySpawner** Klasa ta odpowiedzialna jest za tworzenie instancji **Enemy** co zadany okres. Jeżeli na mapie nie znajdują się żadni przeciwnicy a wszystkie **EnemySpawner** skończyły tworzenie przeciwników, gra jest wygrana przez graczy.

Klasa **Base** główny punkt, którego muszą bronić gracze przed atakami przeciwników. W momencie, gdy jego punkty życia spadną poniżej minimalnej wartości gracze przegrywają.

13 Maja 2025 r.

POLITECHNIKA GDAŃSKA

Sprawozdanie – część trzecia

Przetwarzanie rozproszone

Bartosz Skorowski 197757 | Kamil Raubo 198328 | Jakub Bot 197839

Wstęp

Gra sieciowa została zaimplementowana w języku C++ oraz używa mechanizmu gniazd systemu Linux.

Kod źródłowy wraz z instrukcją uruchomienia znajduje się na githubie: https://github.com/oosiriiss/rozproszone-projekt

Model Komunikacji

Komunikacja oparta jest na gniazdach TCP, a klienci komunikują się z serwerem (i serwer z klientami) za pomocą określonego zbioru komunikatów o ustalonym formacie.

Format komunikatu

Pole	Rozmiar	Opis
Version	4 bajty	Wersja używana w celu wykrycia niezgodności pomiędzy wersjami komunikujących sie aplikacji
Type	2 bajty	Liczba przedstawiająca typ komunikatu, pozwala na odpowiednie odkodowanie przesłanych danych
ContentLength	2 bajty	Długość ciała komunikatu w bajtach.
Timestamp	8 bajtów	Znacznik czasu w milisekundach (milisekundy od 1970-01-01 00:00:00)
Content	Zależny od ContentLength	Dane komunikatu

Rodzaje komunikatów

Komunikaty zdefiniowany są w pliku: **packet.hpp**

W projekcie przyjęto konwencje nazewnictwa, zgodnie z którą wszystkie komunikaty wysyłane przez klient zakończone są sufiksem “Request” a komunikaty wysyłane z serwera do klientów sufiksem “Response”. Większość komunikatów przesyłana jest jako czyste dane binarne (kopia pamięci), jednak nie wszystkie na to pozwalają (gdy np. Zawierają jakiś wskaźnik). Z tego powodu komunikaty mogą nadpisać metodę serializacji i deserializacji (z formatu przesyłowego). A dla klas **Enemy** oraz **Fireball** dodano specjalne klasy DTO (Data Transfer Object), ograniczające ilość przesyłanych danych na temat obiektów do minimum (zawierają tylko kluczowe dane potrzebne do jednoznacznego odtworzenia obiektu).

Legenda typów używanych w poniższej specyfikacji.

- vector2f – dwuwymiarowy wektor (struktura o dwóch zmiennych typu float)
- Direction – Enum zawierający kierunki UP, LEFT, DOWN, RIGHT
- Enemy::DTO – Data transfer object, struktura zawierająca kluczowe pola przeciwnika takie jak: pozycja na mapie, cel na który sie kieruje oraz ilość punktów życia.
- Fireball::DTO – Data transfer object, struktura zawierająca aktualną pozycję obiektu oraz jego kierunek poruszania się.

PlayerDisconnectedResponse

Wysyłane przez serwer do klientów po rozłączeniu się gracza z ID równym <playerID>.

Member	Typ	Opis
playerID	int32_t	Identyfikator gracza, który się rozłączył

JoinLobbyRequest

Wysyłane przez klienta w momencie, gdy dołącza on do poczekalni.

(brak pól)

JoinLobbyResponse

Wysyłane przez serwer do klientów w momencie, gdy nowy gracz dołączy do poczekalni. Zawiera on mapę graczy oraz ich stanów gotowości

Member	Typ	Opis
lobbyPlayers	Map[int32_t,bool]	Mapa graczy w lobby: klucz → ID gracza, wartość → flaga gotowości

LobbyReadyRequst

Wysyłane przez klienta do serwera, gdy zmienia on swój status gotowości w poczekalni na <isReady>

Member	Typ	Opis
isReady	bool	Flaga oznaczająca, czy gracz jest gotowy w lobby

LobbyReadyResponse

Wysyłane przez serwer do klientów w momencie, gdy jakiś gracz o id <playerID> zmieni swój stan gotowości w poczekalni na <isReady>.

Member	Typ	Opis
playerID	int32_t	Identyfikator gracza, którego status gotowości dotyczy
isReady	bool	Flaga gotowości gracza w lobby

StartGameResponse

Wysyłane, gdy wszyscy gracze w poczekalni ogłoszą stan gotowości - jest to sygnał, że gracze mogą przejść do stanu rozgrywki (pętli gry).

(brak pól)

GameReadyRequest

Wysyłane przez klientów zaraz po przejściu do pętli gry. Informuje on serwer o tym, że są oni gotowi do otrzymywani komunikatów i rozpoczęcia rozgrywki.

(brak pól)

GameReadyResponse

Komunikat inicjalizujący grę na kliencie (mapę, pozycje startowe graczy).

Member	Typ	Opis
thisPlayerID	int32_t	ID aktualnego gracza
thisPlayerPos	vector2f	Pozycja aktualnego gracza na mapie
otherID	int32_t	ID drugiego gracza
otherPlayerPos	vector2f	Pozycja drugiego gracza na mapie
map	uint8_t, int8_t[16][16]	Identyfikator mapy oraz dwuwymiarowa tablica zawierająca informacje o typu pola (x,y) (np. Czy jest to zwykła podłoga, ściana, spawner przeciwników, baza, punkt startowy graczy)

PlayerMoveRequest

Wysyłane na serwer przez klienta, gdy się porusza.

Member	Typ	Opis
direction	Direction	Kierunek ruchu wykonywanego przez gracza

PlayerMoveResponse

Wysyłane przez serwer do klientów, gdy gracz o ID <playerID> poruszy się na nową pozycję <newPos>

Member	Typ	Opis
playerID	int32_t	ID gracza, który się poruszył
newPos	vector2f	Nowa pozycja gracza po wykonanym ruchu

EnemyUpdateResponse

Komunikat zawierający dynamiczną listę wymaganych informacji o przeciwnikach w rozgrywce.

Member	Typ	Opis
enemies	std::vector<Enemy::DTO>	Wektor obiektów DTO reprezentujących stan każdego wroga

FireballShotRequest

Komunikat wysyłany przez klienta w momencie, gdy strzeli on kulą ognia.

Member	Typ	Opis
playerID	int32_t	ID gracza, który wystrzelił kulę ognia

fireball	Fireball::DTO	Dane dotyczące wystrzelonej kuli ognia

UpdateFireballsResponse

Komunikat zawierający dynamiczną listę wymaganych informacji o obiektach Fireball w rozgrywce.

Meber	Typ	Opis
fireballs	std::vector<Fireball::DTO>	Wektor obiektów DTO reprezentujących wszystkie kule ognia na mapie

BaseHitResponse

Komunikat informujący, że baza główna została trafiona i posiada teraz <newHealth> punktów życia.

Member	Typ	Opis
newHealth	int	Nowa wartość zdrowia bazy po trafieniu

GameOverResponse

Komunikat wysyłany w momencie zakończenia rozgrywki informujący o końcowym jej wyniku (zwycięstwo lub porażka)

Member	Typ	Opis
isWon	int	Flaga rezultatu gry (np. 1 → wygrana, 0 → przegrana)

Rozgrywka

Sterowanie

- W, A, S, D poruszanie w górę, lewo, dół oraz prawo.
- Lewy przycisk myszy - Strzał przez gracza pociskiem.

W sekcji tej opisane jest przykładowe działanie aplikacji z serwerem i dwoma graczami.

Faza poczekalni

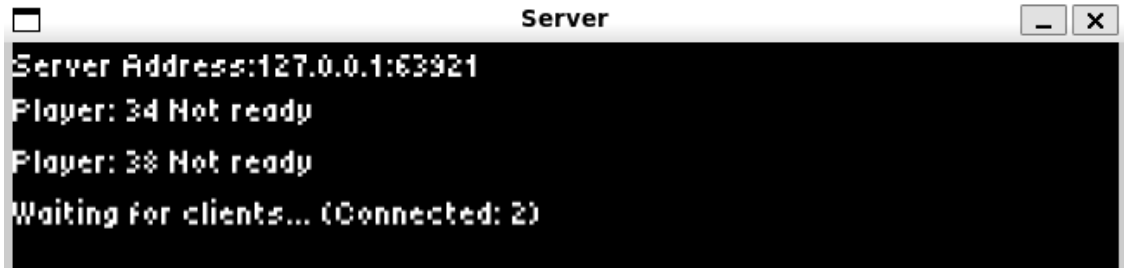
Rysunek 1. Okno serwera po włączeniu aplikacji

Rysunek 2. Okno Klienta po włączeniu aplikacji

Adres serwera ustawiony jest na lokalny adres 127.0.0.1, a port na 63921.

By rozpocząć komunikację najpierw trzeba stworzyć gniazda, co wykonywane jest po naciśnięciu przycisku **Bind** dla okna serwerowego (co jednocześnie tworzy poczekalnie), a następnie **Connect** w przypadku okien klientów (co automatycznie dołącza gracza do poczekalni).

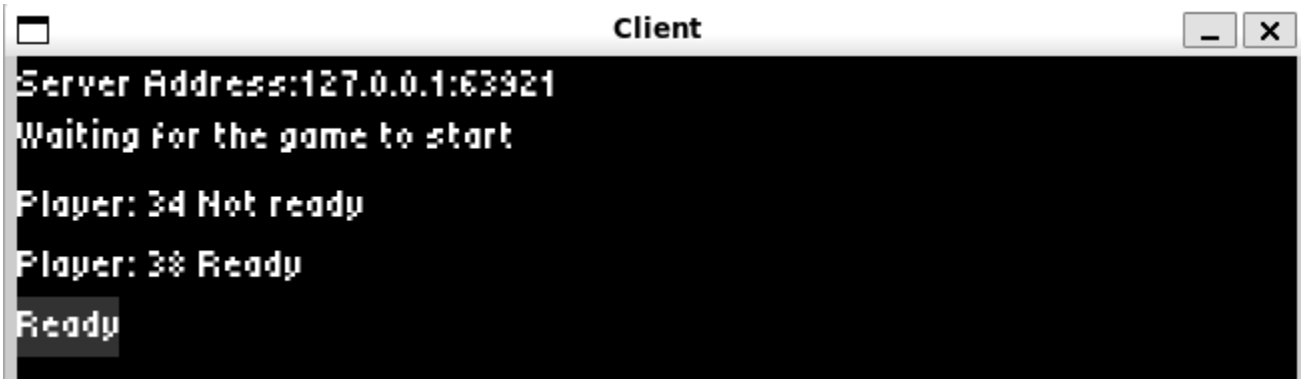
Rysunek 3. Okno serwera po naciśnięciu “Bind” i stworzeniu poczekalni



Rysunek 4. Widok poczekalni po dołączeniu obu graczy

Rysunek 5. Widok poczekalni klienta, po dołączeniu 2 graczy

W poczekalni gracze mogą określić swój stan gotowości przyciskiem **Ready,** a po każdym jego naciśnięciu wysyłają komunikat **LobbyReadyRequest** do serwera, a on rozsyła komunikat **LobbyReadyResponse** do reszty klientów, po czym aktualizują oni swoją lokalną wersję poczeklani.



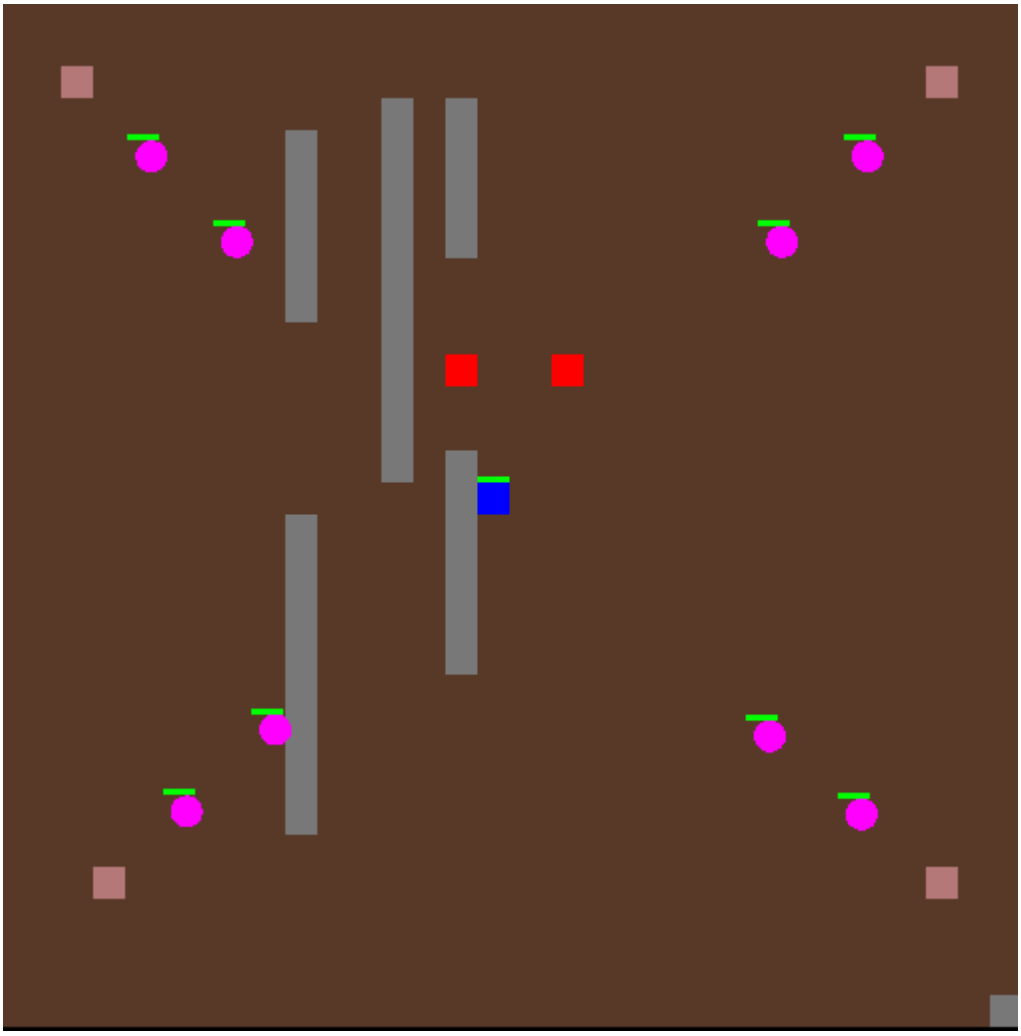
Rysunek 6. Wygląd poczekalni postronie klienta po tym jak drugi gracz ogłosił swoją gotowość.

W momencie, Gdy wszyscy gracze będą gotowi serwer podejmie decyzję o rozpoczęciu gry wysyłając komunikat **StartGameResponse** do klientów, co spowoduje przejście klientów do sceny (ekranu) zawierającego główną pętlę gry.

Po przejściu do tego stanu Klienci wysyłają serwerowi komunikat **GameReadyRequest,** który świadczy, że są oni gotowi na przyjmowanie komunikatów o rozgrywce. Serwer odpowiada tę informację komunikatem **GameReadyResponse,** który inicjalizuje elementy rozgrywki gracza, takie jak mapa i pozycje startowe graczy.

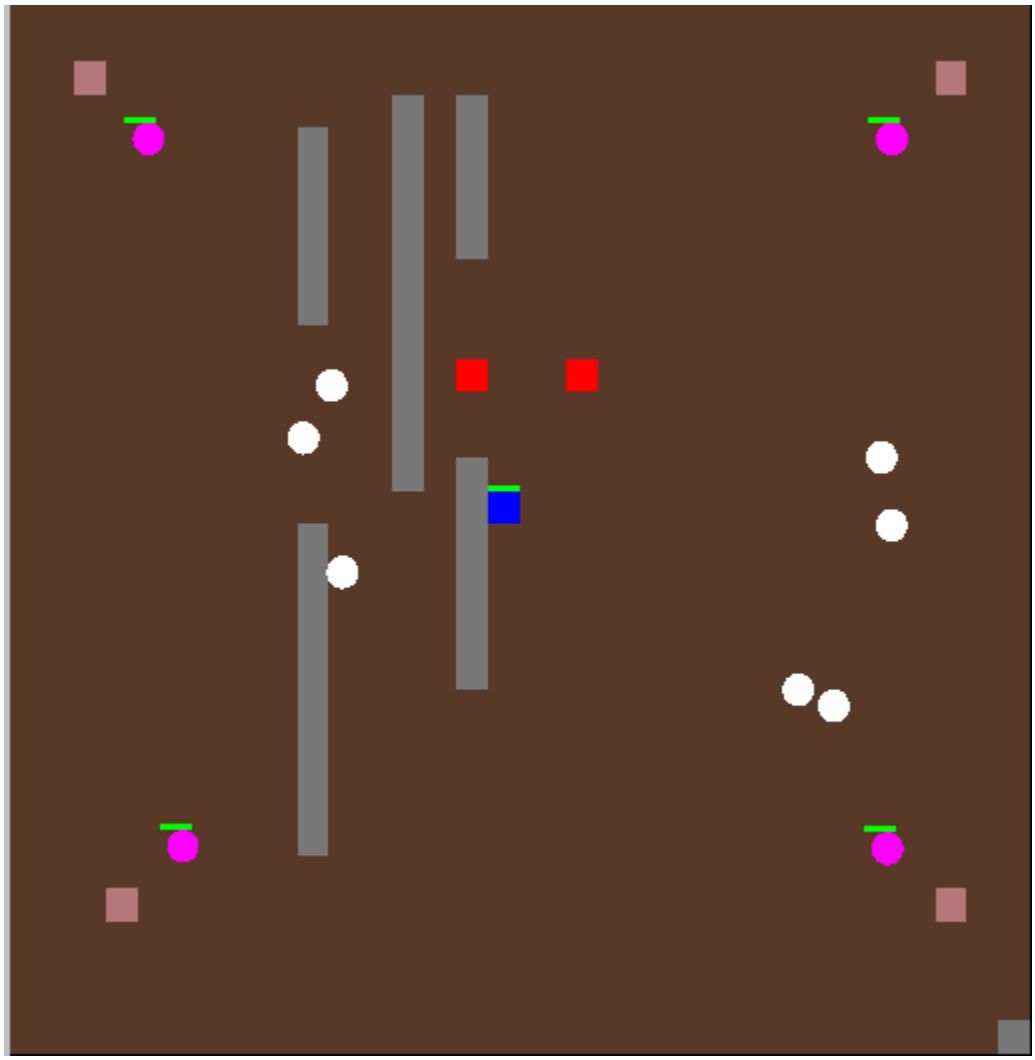
Faza gry

W przypadku zaimplementowanej mapy gracze (Czerwone kwadraty) rozpoczynają rozgrywke w okolicy środka mapy. Baza główna (Granatowy kwadrat) zlokalizowany jest w centrum mapy, a na rogach mapy znajdują się punkty (jasnobrązowe kwadraty), z których pojawiać się będą przeciwnicy (różowe kółka).



Rysunek 7. Przykładowy wygląd rozgrywki

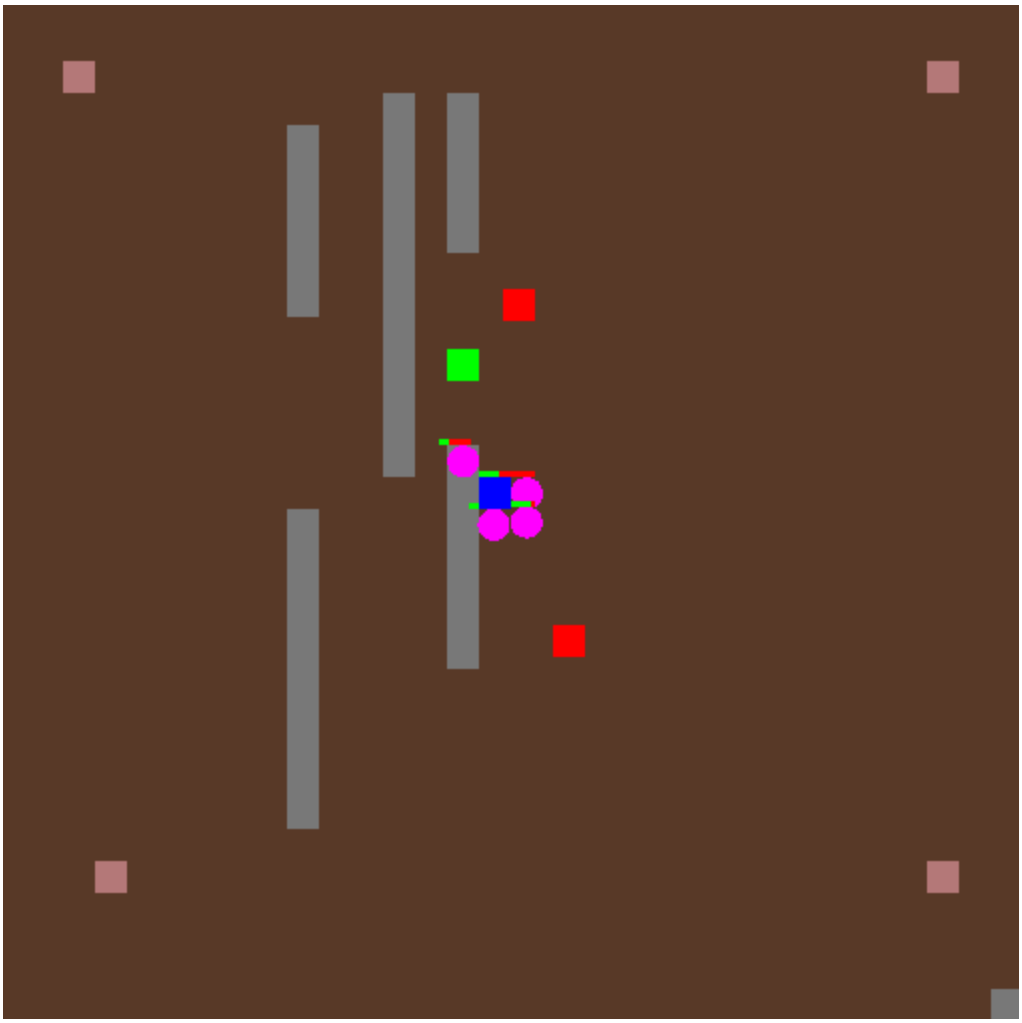
Podczas gry, gracze mogą się poruszać za pomocą klawiszy WASD, wysyłając do serwera komunikaty **PlayerMoveRequest,** a serwer informuje o tych pozostałych klientów wysyłając **PlayerMoveResponse.**



Rysunek 8. Kule wystrzelone przez graczy. Kule po stronie lewej wystrzelone są przez klienta 1, a kule po prawej stronie przez klienta 2

Gracze mogą również strzelać kulami (białe koła) za pomocą lewego przycisku myszy, które będą zadawały obrażenia wrogom. Przy wystrzale kuli ognia, klient wysyła **FireballShotRequest**, a serwer rozsyła to dalej jako **UpdateFireballResponse**. W Przypadku, gdy kula trafi wroga jego pasek życia zmniejsza się, a gdy spadnie od do zera – przeciwnik znika.

Logika przeciwników obliczana jest na serwerze a następnie rozsyłana do graczy co stały okres. Jako **EnemyUpdateResponse.** W lokalnej wersji świata klienci tylko “zgadują” następne pozycje przeciwników prowadząc lokalne symulacje, co minimalizuje efekt zacinania sie przeciwników podczas poruszania, jednak cała logika zadawania obrażen przeciwnikom obsługiwana jest przez serwer.



Rysunek 9. Przedstawiający przeciwników o zmniejszonych paskach zdrowia, którzy dotarli do bazy głównej graczy i zadają jej obrażenia.

Gdy przeciwnik (różowe koło) zada obrażenia bazie głównej (Niebieskiemu kwadratowi) serwer rozsyła komunikat **BaseHitResponse** zawierający zaktualizowaną wartość punktów życia bazy głównej.

Gdy gra zakończy, czyli Gracze pokonają wszystkich przeciwników (zwycięstwo) lub przeciwnicy zniszczą bazę główną graczy (porażka) serwer wysyła komunikat **GameOverResponse,** świadczący o zakończonej rozgrywce, gracze wyświetlają

w konsoli wynik gry (porażka lub zwycięstwo) i wraz z serwerem wracają do poczekalni, gdzie mogą rozegrać grę od nowa.