Czołgi - Wieloosobowa Gra Sieciowa

Przegląd Projektu

Czołgi to wieloosobowa gra czasu rzeczywistego, w której gracze sterują czołgami poruszającymi się po labiryntowym środowisku. Celem gry jest wyeliminowanie innych graczy poprzez trafienie ich pociskami, które mogą odbijać się od ścian. Projekt implementuje rozproszoną architekturę klient-serwer z synchronizacją w czasie rzeczywistym, co czyni go doskonałym przykładem przetwarzania rozproszonego i programowania współbieżnego.

Trzeci Etap Projektu z Przetwarzania Rozproszonego

To repozytorium reprezentuje trzeci etap projektu z kursu przetwarzania rozproszonego. Implementacja koncentruje się na:

1. **Systemie rozproszonym czasu rzeczywistego** - Kompletna architektura klient-serwer z synchronizacją stanu
2. **Zarządzaniu współbieżnością** - Obsługa wielu klientów jednocześnie z operacjami bezpiecznymi dla wątków
3. **Komunikacji sieciowej** - Wykorzystanie protokołu UDP dla aktualizacji stanu gry o niskim opóźnieniu
4. **Mechanizmach synchronizacji** - Implementacja systemu opartego na tickach dla spójnego stanu gry na wszystkich klientach

Architektura

Model Klient-Serwer

Gra wykorzystuje architekturę klient-serwer, gdzie:

- **Serwer**: Utrzymuje autorytatywny stan gry, przetwarza akcje graczy i rozsyła aktualizacje
- **Klienci**: Wysyłają akcje graczy do serwera i renderują stan gry otrzymany z serwera

Komunikacja Sieciowa

- Wykorzystuje protokół UDP dla komunikacji o niskim opóźnieniu
- Implementuje niestandardowy system serializacji wiadomości dla efektywnego przesyłania danych
- Elegancko obsługuje utratę pakietów (korzyść z używania UDP dla gier czasu rzeczywistego)

Implementacja Współbieżności

Współbieżność po Stronie Serwera

Serwer zarządza wieloma operacjami współbieżnymi:

1. **Wątek Pętli Gry**: Aktualizuje stan gry w stałych odstępach czasu (tickach)
2. **Wątek Obsługi Klientów**: Przetwarza przychodzące wiadomości od klientów
3. **Zarządzanie Sekcją Krytyczną**: Zapewnia bezpieczny dla wątków dostęp do współdzielonego stanu gry

Synchronizacja Oparta na Tickach

- Stan gry jest aktualizowany w stałych odstępach czasu zwanych "tickami" (domyślnie: 60 ticków na sekundę)
- Wszystkie akcje graczy są przetwarzane w kontekście bieżącego ticka
- To podejście zapewnia spójne doświadczenie rozgrywki na wszystkich klientach niezależnie od warunków sieciowych

Rozwiązywanie Konfliktów

- Serwer jest ostatecznym autorytetem w rozwiązywaniu konfliktów
- W przypadku jednoczesnych zdarzeń (np. wielu graczy trafionych w tym samym ticku), serwer określa wynik
- Utracone połączenia są elegancko obsługiwane, umożliwiając graczom ponowne połączenie

Funkcje Gry

- Do 4 graczy w jednej grze
- Labiryntowe środowisko ze ścianami
- Pociski, które mogą odbijać się od ścian

- Ruch i strzelanie w czasie rzeczywistym
- System eliminacji po jednym trafieniu

Sterowanie

- **W/S**: Ruch do przodu/do tyłu
- **A/D**: Obrót w lewo/prawo
- **Spacja**: Strzał

Konfiguracja i Użytkowanie

Wymagania

- Python 3.6+
- Biblioteka Pygame

Uruchamianie Serwera

```
python server/server.py
```

Uruchamianie Klienta

```
python client/client.py [nazwa_gracza] [ip_serwera]
```

Wartości domyślne:

- nazwa_gracza: "Player"
- ip_serwera: "localhost"

Struktura Projektu

- **client/** - Implementacja po stronie klienta
- **server/** - Implementacja serwera
- **common/** - Kod współdzielony między klientem a serwerem
  - **game.py** - Główna logika gry
  - **map.py** - Generowanie i zarządzanie mapą

- **player.py** - Implementacja encji gracza
- **bullet.py** - Fizyka pocisków i wykrywanie kolizji
- **network.py** - Warstwa komunikacji sieciowej

Kluczowe Klasy i Funkcje

Klasy NetworkManager i NetworkMessage

Warstwa komunikacji sieciowej jest zaimplementowana przez te dwie klasy:

```
class NetworkMessage:
    """
    Represents a message that can be sent over the network.
    """
    def __init__(self, msg_type, data=None):
        self.msg_type = msg_type
        self.data = data or {}

    def to_bytes(self):
        """Serialize the message to bytes for network transmission"""
        # Serialize data to JSON
        json_data = json.dumps(self.data).encode('utf-8')
        # Create header with message type and data length
        header = struct.pack('!BI', self.msg_type, len(json_data))
        # Return combined header and data
        return header + json_data
```

```
class NetworkManager:
    """
    Manages network communication between clients and server.
    """
    def __init__(self, is_server, host='0.0.0.0', port=12345):
        self.is_server = is_server
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.clients = {}  # Maps client addresses to player IDs

    def send_message(self, message, address=None):
        """Send a message to a specific client or broadcast to all clients"""
        message_bytes = message.to_bytes()

        if self.is_server and address is None:
            # Broadcast to all clients
            for client_address in self.clients:
                self.socket.sendto(message_bytes, client_address)
        else:
            # Send to specific address
            self.socket.sendto(message_bytes, address)
```

Klasa Game

Klasa Game stanowi rdzeń logiki gry, zarządzając stanem gry i przetwarzając akcje graczy:

```
class Game:
    """
    Represents the game state and logic.
    """
    def __init__(self, map_size=(20, 20), max_players=4, tick_rate=60):
        self.map = Map(map_size[0], map_size[1])
        self.players = []
        self.bullets = []
        self.defeated_players = []
        self.is_running = False
        self.max_players = max_players
        self.tick_rate = tick_rate
        self.last_tick_time = 0

    def update(self):
        """
        Update the game state for one tick.
        Returns True if the game is still running, False if it has ended.
        """
        # Update bullets
        active_bullets = []
        for bullet in self.bullets:
            if bullet.update(self.map, self.players):
                active_bullets.append(bullet)
        self.bullets = active_bullets

        # Update players
        for player in self.players:
            player.update()

        # Check if game should end (only one player left alive)
        alive_players = [p for p in self.players if p.is_alive]
        if len(alive_players) <= 1 and self.is_running:
            self.end_game()
            return False

        return self.is_running
```

Klasa Player

Klasa Player reprezentuje gracza w grze:

```
class Player:
    """
    Represents a player in the game.
    """
    def __init__(self, name, position=(0, 0), direction=(1, 0), ip_address=None):
        self.name = name
        self.position = position
        self.direction = direction
        self.ip_address = ip_address
        self.is_alive = True
        self.bullets = []
        self.rotation_speed = 0.1  # Radians per tick
        self.movement_speed = 0.05  # Cells per tick

    def fire_bullet(self):
        """
        Fire a bullet in the direction the player is facing.
        Returns the created bullet.
        """
        bullet = Bullet(self.position, self.direction, self)
        return bullet

    def move_forward(self, game_map):
        """
        Move the player forward in the direction they are facing.
        """
        dx, dy = self.direction
        new_x = self.position[0] + dx * self.movement_speed
        new_y = self.position[1] + dy * self.movement_speed

        # Check for wall collision
        if game_map.is_position_valid(round(new_x), round(new_y)):
            self.position = (new_x, new_y)
```

Klasa Bullet

Klasa Bullet obsługuje fizykę pocisków i wykrywanie kolizji:

```python
class Bullet:
    """
    Represents a bullet fired by a player.
    """
    def __init__(self, position, direction, owner=None, max_bounces=15, speed=0.09):
        self.position = position
        self.direction = direction
        self.owner = owner
        self.hit = False
        self.bounces = 0
        self.max_bounces = max_bounces
        self.speed = speed
        self.life_time = 0

    def update(self, game_map, players):
        """
        Update the bullet's position and check for collisions.
        Returns True if the bullet is still active, False if it should be removed.
        """
        # Move the bullet
        new_x = self.position[0] + self.direction[0] * self.speed
        new_y = self.position[1] + self.direction[1] * self.speed

        # Check for wall collision and handle bounces
        if not game_map.is_position_valid(round(new_x), round(new_y)):
            # Handle bounce logic...
            self.bounces += 1
            if self.bounces >= self.max_bounces:
                return False

        # Check for player collision
        for player in players:
            if player.is_alive and self.collision_with_player(player):
                self.hit = True
                player.is_alive = False
                return False

        return True
```

Pętla Gry Serwera

Pętla gry serwera jest odpowiedzialna za aktualizację stanu gry w stałych odstępach czasu:

```
def _game_loop(self):
    """
    The main game loop that updates the game state at a fixed rate.
    """
    last_tick_time = time.time()

    while self.running:
        current_time = time.time()
        elapsed = current_time - last_tick_time

        if elapsed >= self.tick_interval:
            # Update game state
            if self.game.is_running:
                game_running = self.game.update()

                # Send game state to all clients
                state_message = NetworkMessage(MSG_TYPE_STATE, self.game.get_state())
                self.network.send_message(state_message)

                # If the game just ended, send an END message
                if not game_running:
                    # Get the winner and send end message
                    winner = None
                    if self.game.defeated_players:
                        winner = self.game.defeated_players[-1].name

                    end_message = NetworkMessage(MSG_TYPE_END, {
                        'winner': winner
                    })
                    self.network.send_message(end_message)

            last_tick_time = current_time
```

Najważniejsze Aspekty Techniczne

1. **Zarządzanie Stanem Rozproszonym**: Serwer utrzymuje autorytatywny stan gry, podczas gdy klienci utrzymują zsynchronizowaną lokalną kopię
2. **Przetwarzanie Współbieżne**: Wiele wątków obsługuje aktualizacje gry i komunikację z klientami jednocześnie
3. **Synchronizacja w Czasie Rzeczywistym**: System ticków zapewnia, że wszyscy klienci widzą spójny stan gry

4. **Odporność na Błędy**: System elegancko obsługuje problemy z siecią i rozłączenia klientów