Konspekt - Gra multiplayer typu shooter

Miłosz Szumiec Maria Kowalczyk

4 maja 2020

1 Wprowadzenie

W ramach projektu przygotowaliśmy grę multiplayer typu shooter dla dwóch graczy, z możliwością ewentualnego dalszego rozszerzania ich liczby. Program napisaliśmy w pythonie korzystając z modułów takich jak pygame, socket, pickle czy thread.

Cel projektu:

- zapoznanie się z modułami pozwalającymi na zdalną synchronizację wielu aplikacji
- zapoznanie się z modułem pygame pozwalającym na tworzenie responsywnych aplikacji graficznych
- zmierzenie się z wyzwaniami jakie stawia synchronizacja dwóch gier bazujących na bardzo dużej liczbie atrybutów

Przedmiot demonstracji:

- przybliżenie działania modułu socket i podstawowego zastosowania modułu pickle
- przedstawienie ogólnego zarysu działania modułu pygame na przykładzie wcześniej stworzonej gry

Zakres demonstracji:

- zdobytą wiedzę można użyć w przyszłych projektach, wymagających zastosowania moduły pygame, bądź połączenia ze sobą większej liczby aplikacji
- ograniczeniem jest fakt, że musimy mieć bezpośrednie połączenie po ip (serwer - klient), co często uniemożliwia protokół NAT stosowany w ruterach
  - można to obejść port-forwardingiem (nie do końca bezpieczne rozwiązanie) lub zastosowaniem wirtualnych sieci lokalnych

2 Część Teoretyczna

2.1 Komunikacja między aplikacjami

2.1.1 Serwer

W celu osiągnięcia płynnej wymiany danych pomiędzy graczami, zdecydowaliśmy się na zastosowanie modułu socket do pythona. Pozwala on na nawiązanie połączenia między dwoma aplikacjami przy pomocy zdefiniowanych przez nas protokołów. W naszym przypadku użyliśmy połączenia TCP po IPv4. IPv4 wybraliśmy ze względu na prostotę zapisywania adresów, a TCP ze względu na połączeniowość protokołu. Chodziło nam o osiągnięcie jednorazowego, ciągłego połączenia między aplikacją kliencką a serwerową oraz o pewność, że dane dotrą do drugiego punktu w całości. Warto jednak zauważyć, że moduł socket pozwala również na inne rodzaje połączeń. Definiujemy to podając odpowiednie argumenty podczas tworzenia socketa.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
AF_INET -> IPv4
SOCK_STREAM -> TCP , SOCK_DGRAM -> UDP
```

Po stworzeniu socketa musimy przypisać mu odpowiedni adres i port, na którym będzie pracował, co osiągamy funkcją bind(), wykorzystującą tuple (adres, port) jako argument. Następnie wywołujemy funkcję listen(), która pozwala serwerowi na akceptowanie nowych połączeń.

```
s.bind((adres, port))
s.listen()
```

Po stworzeniu takiego socketa nie zostaje nam nic innego jak rozpocząć przyjmowanie przychodzących połączeń. Osiągamy to poprzez stworzenie nieskończonej pętli, w której bedziemy akceptować próby połączenia się z nami, przy pomocy funkcji accept(), zwracającej obiekt będący samym połączeniem, jak i adres klienta. Po otrzymaniu takich danych rozpoczynamy nowy wątek na serwerze który będzie odpowiedzialny za obsługiwanie danego klienta. Takie rozwiązanie pozwala nam obsługiwać więcej niż jednego klienta jednocześnie i jest konieczne do osiągniecia pożądanych przez nas rezultatów. Nowy wątek tworzymy przy pomocy funkcji start_new_thread(), biorącej za argumenty funkcję, którą ma wykonywać i argumenty do tej funkcji, w postaci obiektu tuple. W naszym przypadku do wątku będziemy przekazywać połączenie z klientem i jego indeks.

```
currentPlayer = 0
while True:
    conn, addr = s.accept()
    start_new_thread(threaded_client, (conn, currentPlayer))
    currentPlayer += 1
```

Zanim przejdziemy do samej wymiany danych pomiędzy klientem a serwerem, warto zauważyć, że serwer domyślnie musi przechowywać dane wszystkich klientów (w naszym przypadku). Dlatego też przed rozpoczęciem pętli tworzymy na serwerze obiekt gry, którą będzie obsługiwał i pobieramy z niej wszystkie argumenty potrzebne do zsynchronizowania ze sobą graczy. Początkowe przesłanie całego obiektu gry do klientów nie jest możliwe przez ograniczenie jakie niesie za sobą moduł socket. Nie pozwala on bowiem na wysyłanie obiektów pygame'a zawierających sprite'y. W celu obejścia tego problemu wysyłamy jedynie atrybuty, a same obiekty tworzymy bądź aktualizujemy już u klientów lokalnie.

```
*screenshot zapisu danych wraz z wytłumaczeniem*
```

Mając za sobą przygotowane łączenie się i wyjściowe dane, możemy przejść do obsługi pojedynczych klientów. W funkcji threaded_client() przed rozpoczęciem ciągłej wymiany wysyłamy jeden pakiet danych zawierający base (atrybuty dla każdego gracza), player (indeks gracza na serwerze) i g.boxes (listę wszystkich obiektów do wygenerowania u klienta) (przede wszystkim skrzyń).

```
def threaded_client(conn, player):
    conn.sendall(pickle.dumps([base, player, g.boxes]))
```

Jak widać na przykładzie przesyłamy klientowi obiekt pickle.dumps(). Funkcja ta z modułu pickle odpowiedzialna jest za odpowiednie zapakowanie danych, aby można je było przesłać. W dalszej części rozpakowywać takie dane będziemy przy użyciu funkcji pickle.loads(). W zadaniu praktycznym zrezygnowaliśmy z zastosowania modułu pickle w celu pokazania, że wysyłanie wartości musi odbywać się w poprawnym formacie (bitowym). Odpowiedni format możemy osiągnąć na przykład poprzez zastosowanie funkcji str.encode(), jak na przykładzie poniżej. Ograniczeniem takiej metody jest możliwość wysyłania jedynie stringów, co nie sprawdzi się przy wysyłaniu wielkich zagnieżdżonych list i słowników.

```
conn.sendall(str.encode('message'))
```

Idąc dalej wkraczamy w kolejną nieskończoną pętlę, dedykowaną tym razem pojedynczemu klientowi. Na samym początku odbieramy dane od klienta przy pomocy funkcji recv(), biorącej za argument bufor stanowiący maksymalną ilość danych w jednym pakiecie. Poniżej załączam dwa przykłady zastosowania tej funkcji, odwołujących się do dwóch poprzednich przykładów zastosowania funkcji sendall().

```
playerData = pickle.loads(conn.recv(8192))  //przykład z gry
info = conn.recv(2048).decode() //przykład z zadania praktycznego
```

Następnie przystępujemy do odpowiedniej obróki tych danych, opierającej się na zaktualizowaniu informacji przechowywanych na serwerze i wysłaniu klientowi informacji o innych użytkowniach, którzy w tym samym czasie dokonują identycznych wymian. Dodatkowo stosujemy instrukcję warunkową, która w razie braku przychodzących informacji zamknie połączenie przy pomocy funkcji close().

`*screenshot kodu z werbalnym wytłumaczeniem*`

Tak właśnie prezentuje się całe działanie serwera. Warto jednak jeszcze zauważyć że do wysyłania możemy używać zarówno funkcji send() jak i sendall(). Różnica między nimi polega przede wszystkim na fakcie, że send() jest niskopoziomową metodą na bazie C i nie gwarantuje że wszystkie pliki przesłane zostaną w całości, a sendall() jest wysokopoziomową metodą wywołującą send() do czasu aż wszystko zostanie przesłane lub w innym razie zwracającą błąd. Dla naszego połączenia TCP zdecydowaliśmy się więc używać sendall().

2.1.2 Klient

Po stronie klienta przeprowadzane przez nas operacje będą analogiczne, ale w celu uproszczenia kodu funkcję stricte socketowe przenieślimy do klasy Network. Jedyną nową zastosowaną tu funkcją jest connect(), która za argument przyjmuje obiekt tuple zawierający adres i port.

`*screenshot i omówienie*`

W samym pliku klienckim zostaje nam więc jedynie stworzyć obiekt Network, który automatycznie połączy się z serwerem. Następnie rozpakowujemy pakiet z początkowymi danymi, zawierający wszystkich graczy (w tym nas) i przechodzimy do aktualizacji lokalnych zmiennych, w zależności od tego którym graczem jesteśmy.

```
n = Network()
base = n.getP()
data = base[0]
player = base[1]
boxes = base[2]
self.gameplay.c_f = player  # ustawiamy id gracza
self.gameplay.current_focus = self.gameplay.players[player]
self.gameplay.sync_windows(data)
self.gameplay.generate_obst_from_data(boxes)
```

Po zsynchronizowaniu się z serwerem nie zostaje nam nic innego jak wejść do nieskończonej pętli odpowiadającej zarówno za wydarzenia w grze, jak i komunikację z serwerem, a co za tym idzie innymi graczami. Na początku każdej takiej pętli pakujemy wszystkie dane lokalne (dotyczące nas). W przypadku gry są to informacje o naszym graczu, wystrzelone pociski i skrzynie, z którymi weszliśmy w interakcję. Po zapakowaniu czyścimy lokalne tablice przechowujące skrzynie i pociski, bo są to jednorazowe informacje i nie chcemy przesyłać ich w nieskończoność.

```
self.gameplay.pickle_player(self.gameplay.c_f)
p1 = [self.gameplay.allPlayersToPickle[player],
self.gameplay.bulletsToPickle, self.gameplay.boxes]

self.gameplay.boxes = []
self.gameplay.bulletsToPickle = []
p2 = n.send(p1)
self.gameplay.update(p2)
```

Po wymianie aktualizujemy swoją grę o informacje wysłane przez pozostałych graczy i cykl zatacza koło.

2.2 Pygame

Tworzenie gier znacznie ułatwia biblioteka pygame i to głównie dlatego zdecydowaliśmy się na język python. W dzisiejszej prezentacji postanowiliśmy przybliżyć Wam podstawy tego modułu.

2.2.1 Podstawowy szkielet każdej aplikacji pygame

Pygame wykorzystuje bibliotekę SDL, dzięki której dostarcza modułów pozwalających przykładowo na wyświetlanie grafiki, odtwarzanie dźwięków, śledzenie czasu, obsługę myszy i joysticka. Używając pygame możemy pisać gry, które będą działały bez zmian na dowolnej platformie obsługiwanej przez SDL (Windows, Linux, MacOS itd.)

- **Importujemy moduł i tworzymy wyświetlacz** - pierwszym krokiem jest zaimportowanie naszego modułu oraz zainicjalizowanie pygame'owych funkcji poleceniem pygame.init(). Następnie przechodzimy do stworzenia wyświetlacza. Zmienna screen, przechowuje obiekt pygame.Surface, wywołany przez pygame.display.setmode. Jak łatwo się domyślić, krotka w nawiasie przekazuje wymiar okna, które otworzy się po wywołaniu programu, a kolejny wers pozwala określić jego tytuł. Warto tutaj zwrócić uwagę właśnie na fakt, że jest to krotka, a nie dwa integery.
- **Pętla gry** - to pętla nieskończona, która zajmuje się obsługą zdarzeń, aktualizacją ekranu oraz obsługą wszystkich elementów w trakcie działania programu (np. przesuwaniem obiektów, rysowaniem obiektów na ekranie,

itd.). Zdarzenie rozumiemy jako działanie użytkownika, takie jak poruszenie myszą lub naciśnięcie klawisza. Pygame zarejestruje wszystkie te zdarzenia w kolejce, którą otrzymamy przez wywołanie pygame.event.get(). Możemy iterować i sprawdzać, czy istnieje zdarzenie, którym chcielibyśmy się zająć. Przykładowo sprawdzamy czy użytkownik nacisnął klawisz. W takim przypadku to zdarzenie ma pewnien klucz atrybutu, którym możemy sprawdzić co reprezentuje. Widzimy pętle, która pobiera kolejne zdarzenia i jesli kliknieto w przycisk zamykajacy okno to konczy program. Polecenie pygame.quit() zamyka wszystkie podmoduły.

- **Tworzenie obiektu** - Kolejnym krokiem będzie umieszczenie na planszy obrazka, który chcemy zanimować. W tym wypadku będzie to nasza postać z gry. Korzystamy tutaj z screen.blit(). Blit służy właśnie do rysowania jednych obiektów Surface na drugich. Użyty wcześniej kod pygame.image.load() tworzy nowy obiekt tego typu, oddzielony od właściwej planszy, dlatego konieczne jest jeszcze ich połączenie. Współrzędne określają położenie animacji na środku ekranu.

- **Mechanizm poruszania się postaci** - korzystamy z funkcji pygame.key.get pressed(), która dostarcza przez cały czas informację, że dany klawisz jest wciśnięty. Wykorzystuje ona do tego tablicę z wartościami True/False. I tak, jeżeli prawdą jest, że klawisz A jest wciśnięty to obiekt porusza się w lewo z szybkością taką jak w zmiennej speed. Mechanizm powtarza się dla klawiszy W, S, D (czyli poruszanie się w górę, w dół i w prawo).

- **Rysowanie ruchu postaci** - funkcją fill - czyścimy ekran, poprzez wypełnienie go czarnym kolorem, funkcją draw - rysujemy naszego gracza, funkcją flip - zamieniamy miejscami. I tak w kółko. Dzięki temu unikamy efektu "śladu" i mamy płynnie poruszającego się gracza za pomocą klawiszy WSAD.

2.2.2 Śledzenie kursora myszki

W związku z ograniczającym nas czasem trwania demonstracji, postanowiliśmy pokazać jak w łatwy sposób można zaimplementować śledzenie kursora myszki przez gracza. Jest to rzecz, która z pewnością przyda nam się w przyszłości przy tworzeniu większości gier zręcznościowych. Po załadowaniu pliku do zmiennej img tworzymy kopię naszego obrazu, korzystając z funkcji copy. Następnie dzięki funkcji **get rect** uzyskujemy kwadrat z naszego obrazka. Kolejnym krokiem jest zdefiniowanie funkcji rotate, w której obracamy o podany kąt nasz obrazek. Kąt uzyskujemy w następujący sposób. Do zmiennych mouseX i mouseY przypisujemy aktualną pozycję kursora myszki. Następnie obliczamy różnicę pomiędzy pozycją myszki i gracza. W kolejnym kroku określamy kąt pomiędzy w/w punktami, korzystając z funkcji atan2. I tutaj należy pamiętać, że wynik otrzymujemy w radianach, dlatego w dalszym kroku musimy skonwertować go na stopnie. Wynik funkcji atan2 jest także z zakresu od -180 do 180, a my potrzebujemy do funkcji rotate zakres od 0 do 360, dlatego musimy dodać do funkcji wartość

180. Tym sposobem uzyskujemy gracza, który płynnie się porusza i śledzi kursor myszki.

3 Część praktyczna

3.1 Pytania

1. Jaki argument odpowiada za połączenie TCP?
   (a) AF_INET
   (b) AF_INET6
   (c) SOCK_DGRAM
   (d) SOCK_STREAM
2. Chcąc wysłać obiekt x = [1, 2, 3] użyjemy komendy:
   (a) conn.sendall(str.encode(x))
   (b) conn.sendall(pickle.dumps(x))
3. Mając obiekt client = socket.socket(socket.AF_INET, socket.SOCK_STREAM), napisz, jaką komendą połączysz się z serwerem o adresie 192.168.2.1 na porcie 5525.
4. Napisz jaki wynik zwraca funcja atan2 i dlaczego dodajemy do niego wartość 180.
5. Do czego służy funkcja blit?
   (a) tworzy kopię grafiki i uzyskuje z niej kwadrat
   (b) przypisuje grafikę do określonego miejsca ekranu
   (c) obraca grafikę o podany kąt

3.2 Praca na kodzie

1. Otwórz plik ZADANKO_1.py i postępuj zgodnie z instrukcjami. Twoim zadaniem jest napisanie prostego przykładowego serwera.
2. Otwórz plik ZADANKO_2.py i postępuj zgodnie z instrukcjami. W tym zadaniu tworzysz prostą aplikację kliencką. Po wykonaniu wszystkich instrukcji przeklej całość do pliku ZADANKO_2.2.py, a następnie uruchom kolejno ZADANKO_1.py , ZADANKO_2.py i ZADANKO_2.2.py.
   Miej na uwadze, że dla uproszczenia serwera każdy kolejny użytkownik przyjmuje kolejny indeks, a komunikacja przewidziana jest dla dwóch pierwszych użytkowników. Przy każdym teście restartuj serwer!
3. Otwórz plik ZADANKO_3.py i postępuj zgodnie z instrukcjami. Twoim zadaniem jest napisanie bardzo prostej aplikacji, korzystając z moduły pygame.

4 Podsumowanie

- Przedstawiliśmy jak w sposób podstawowy wdrożyć do programu bibliotekę socket i pickle, oraz jakie niosą ze sobą ograniczenia.
- Pokazaliśmy możliwości, jakie niesie ze sobą moduł pygame, przy którym głównym ograniczeniem potencjalnego użytkownika jest wyobraźnia.
- Po wysłuchaniu naszej demonstarcji oraz przerobieniu ćwiczeń, będziecie w stanie stworzyć podstawowy szkielet gry typu shooter, który w przyszłości będziecie mogli rozszerzać.

5 Bibliografia

- https://docs.python.org/3/library/socket.html
- https://docs.python.org/2/library/socket.html
- https://docs.python.org/3/howto/sockets.html
- https://docs.python.org/2/library/thread.html
- https://docs.python.org/3/library/pickle.html
- https://stackoverflow.com/questions/34252273/what-is-the-difference-between-socket-send-and-socket-sendall
- https://techwithtim.net/tutorials/python-online-game-tutorial/connecting-multiple-clients/
- https://www.pygame.org/news
- "Game Programming" by Andy Harris 2007
- https://www.youtube.com/watch?v=PVY46hUp2EM