Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедіа факультету інформатики

РОЗРОБКА ГРИ З ВИКОРИСТАННЯМ ФРЕЙМВОРКУ UNITY

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення”

Керівник курсової роботи

Борозенний С.О

Виконав студент

Будаєв А.Ю.

Київ 2022

Анотація

У роботі розглянуті проблеми та особливості клієнт-серверної архітектури, протоколів обміну даними та шаблонів програмування при розробці багатокористувацьких ігор. Також розглянута практична реалізація проекту гри на прикладі багатокористувацької онлайн-гри.

Вступ

На сьогодні індустрія ігор набуває все більших обертів завдяки розвитку інноваційних технологій. З кожним роком кількість геймерів зростає як серед чоловіків, так і серед жінок.

Техніка стає все більш доступною, а розробники докладають чимало зусиль, щоб зробити свої продукти більш яскравими та реалістичними. Ми є свідками появи ігор з віртуальними реальностями, де людям цікавіше знаходитися, ніж у реальному житті.

Однак, разом з цим механіки стають складнішими, кількість гравців зростає, а тому навантаження на сервери стає більшим. Це потребує здатності підтримувати великої кількості одночасних з’єднань. Саме тому постає питання щодо максимальної оптимізації ігор, правильному виборі шаблонів як у програмуванні, так і архітектурі.

Subnautica, Cities Skylines, Hearthstone - яскраві приклади популярних онлайн-ігор, які написані за допомогою фреймворку Unity. Їхня аудиторія сягає мільйонів гравців, які надихаються масштабністю, якістю графіки та складністю ігрового процесу. Наведені продукти наштовхують на думку, що Unity - один з кращих виборів для написання багатокористувацької гри.

Зміст

Розділ 1. Особливості клієнт-серверної взаємодії

1.1 Загальна логіка взаємодії клієнта з сервером

Гра представлена у такий спосіб, що клієнт посилає дані про дії гравця на сервер, сервер приймає рішення, перевіряє, чи дія може бути зроблена. Після цього розсилає всім іншим гравцям відомості про дію, яку здійснив гравець.

Треба відштовхуватись від того, що постійно існують ризики злому клієнта. Щоб запобігти цій небезпеці необхідно звести активність клієнта до того, щоб він лише пересилав дані та здійснював перетворення локального світу на основі отриманих даних і у ніякий спосіб не змінював дані про світ на сервері. Тобто всі повідомлення, надіслані з клієнту мають бути перевірені, наприклад:

- якщо гравець натиснув клавішу «Вліво» - клієнт має опрацювати введення, може перемістити гравця локально і передати дані про те, що гравець перемістився, на сервер. Далі останній має перевірити, чи дійсно гравець може бути переміщений за таким вектором (адже на його шляху може бути перепона або ж швидкість гравця перевищує реальну). Після всіх перевірок на можливість дії руху сервер має оновити в себе стан про положення гравця та надіслати всім іншим гравцям, що певний гравець зсунувся на заданий вектор. Клієнти інших гравців мають зсунути в себе модель цього гравця на заданий вектор.

- якщо гравець вистрілив у іншого гравця, його клієнт може включити локально анімацію пострілу та передати інформацію про нього на сервер, а саме промінь пострілу з початковою позицією та направленням. Сервер має перевірити, чи дійсно гравець знаходився там, де починався промінь і визначити, який гравець знаходився на шляху променю. Після того опрацювати нанесену шкоду за своєю логікою, оновити свій стан відповідно до цього і повідомити інших

гравців, що гравець А вистрілив в гравця Б з певною шкодою. Клієнти інших гравців мають опрацювати це, змінивши шкалу здоров’я гравця Б у себе локально.

Таким чином, якщо клієнт буде зламаний, то всі нечесні дії (надто висока швидкість, проходження крізь стіни, аномальна шкода від пострілу) буде відбуватись лише у нього локально і не вплине на реальний стан сервера.

1.2 Транспортний протокол передачі даних

Під час розробки онлайн-ігор дуже важливо обрати відповідний транспортний протокол передачі даних між клієнтом та сервером, а також протокол рівня застосунку.

Щодо вибору транспортного протоколу є три варіанти – TCP, UDP, а також створення власного протоколу на основі UDP. Необхідно зважити переваги та недоліки кожного з наведених варіантів.

TCP гарантує доставку пакетів IP між клієнтом та сервером, а також послідовність їхнього надходження. Однак, це збільшує втрату часу та навантаження на мережу, адже TCP отримує свої переваги завдяки значно більшому, ніж у UDP, розміру супроводжувальних даних і, як наслідок, більшому розміру сегментів.

Водночас UDP є тонкою надбудовою над протоколом IP, що дає змогу значно знизити розмір пакету і відповідно навантаження на мережу, а також швидкість передачі даних. При цьому UDP не гарантує їхню доставку до кінцевого адресата.

Отже, під час вибору з цих двох протоколів треба виходити з того, наскільки дані, які ми передаємо, потребують гарантованої доставки. Наприклад, якщо дані критичні для ігрового процесу та їхня втрата призведе до видалення певної ігрової події, то слід обирати TCP. Якщо ж дані мають потоковий характер (наприклад, онлайн-трансляція всередині гри) і втрата

їхньої частини не така важлива порівняно з перевагою від швидкістю передачі - вибір однозначно за UDP.

Варіант, який передбачає створення власного транспортного протоколу, вимагає додаткового часу та витрат з боку розробника, але дає певну гнучкість. У разі, якщо ігровий процес перш за все потребує високої швидкості передачі даних, а не цілісності їхньої передачі даних, тоді корисно застосовувати власну надбудову над UDP і контролювати транспортний рівень вручну, враховуючи ті необхідності, які потребує ігровий процес.

1.3 Протокол передачі даних рівня застосунку

Після вибору транспортного протоку не менш важливим є протокол передачі даних рівня застосунку, а саме те, як клієнт і сервер будуть кодувати дані для передачі за обраним транспортним протоколу.

На перший погляд, можна обрати JSON або XML. Такий підхід є найпростішим для розробника, адже більшість мов програмування мають бібліотеки для кодування даних у цих форматах з подальшою їх серіалізацією у байтові послідовності. Але треба розуміти, що дані, які несуть інформацію про події, будуть доповнені супровідною інформацією вищеперерахованих форматів. Вони створені для легкого прочитання людиною, а оскільки передбачається, що ці дані будуть прочитані клієнтом або сервером, то такий підхід несе в собі зайві надбудови у розмірі пакетів та втрату продуктивності.

Очевидний вибір – створення власного байтового кодування інформації, для детермінованої кількості різновидів повідомлень, якими обмінюються між собою клієнт та сервер, а також модулі кодування/декодування мережевих пакетів на обох сторонах цього процесу. Завдяки цьому можна позбавитись додаткової надбудови JSON або XML.

1.4 Підтримка багатьох з’єднань одночасно

Перед розробниками постійно постає питання, яким чином серверу підтримувати багато з’єднань з гравцями та обмінюватись даними у двонаправленому режимі.

До прикладу, протокол HTTP є однонаправленим, тобто сервер очікує нове з’єднання від клієнта і надсилає дані на його запит. Для сайтів це не є проблема, оскільки для отримання веб-сторінки цього досить. Хоча для отримання оновленої інформації необхідно оновити сторінку, сервер не буде її досилати без відповідного запиту клієнта. Однак, чи можна застосовувати цей протокол у онлайн-грі, у ігровому світі якої кожну секунду можуть відбуватися тисячі змін? Для цього клієнтам треба у фоновому режимі постійно надсилати запити про зміни на сервер, при цьому кожен раз буде створюватися нове HTTP з’єднання, що займає час. Враховуючи те, що частота змін може перевищувати кілька десятків на секунду, то про жодну високу швидкість обміну даними не може бути й мови. Якщо знизити частоту запитів задля меншого навантаження на мережу, то гравець буде отримувати дані з певним запізненням, що робить динамічний ігровий процес неможливим. З іншого боку, за відсутності змін у ігровому світі клієнт буде надсилати марні запити, тим самим перевантажуючи мережу.

Вирішенням цієї проблеми є WebSocket – протокол, який є надбудовою над TCP та забезпечує двонаправлений зв’язок між клієнтом та сервером без додаткових HTTP повідомлень. Він ідеально підходить для створення онлайн-ігор. Ініціюється він у такий спосіб: клієнт надсилає такий HTTP запит:

```
GET /somepath HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: somesite.com
```

Origin: http://somesite.com

Сервер відповідає:

HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
WebSocket-Origin: http://somesite.com
WebSocket-Location: ws:// somesite.com/demo

Надалі клієнт та сервер залишають з'єднання відкритим і можуть обмінюватися даними у обидві сторони у реальному часі без додаткових часових втрат та HTTP повідомлень і рукостискань.

1.5 Конкурентний доступ до даних від багатьох з'єднань

Для збільшення швидкості обробки даних очевидно, що зчитування та обробка даних зі з'єднання з клієнтами мають відбуватися у паралельному потоці. Але всі з'єднання так чи інакше впираються у «вузьке місце» у вигляді даних, що безпосередньо зберігають стан гри. Таким чином, необхідно забезпечити конкурентний доступ до цих даних від потоків обробки з'єднань відповідними програмними засобами мови, якою написана серверна частина. Проблеми, що можуть виникнути, якщо цього не зробити:

- «Брудне зчитування» - припустімо, що з ініціативи одного потоку гра почне змінювати свій стан, потім виникне помилка і стан буде змінений назад. Але під час цього інший потік може отримати дані до зворотних змін.
- Втрата оновлень - оновлення стану гри з одного потоку буде перезаписано оновленням з іншого потоку.

- Неоднакове зчитування – в рамках однієї обробки стану гри з одного потоку один і той же показник при різних зчитуваннях може різнитися, якщо між зчитуванням інший потік перезаписав цей показник.
- Фантомні записи – в рамках однієї обробки стану гри один потік намагається зчитати двічі один показник, але при цьому між цими зчитуваннями інший потік перезаписав цей показник і при другому зчитуванні перший потік отримає помилку.

1.6 Оптимізації

Окрім вищеперерахованих оптимізацій щодо мережевих пакетів даних, варто виділити такі, що дозволяють зменшити кількість даних рівня застосунку, а також складність їхньої обробки сервером.

По-перше, непотрібно передавати інформацію про світ, яка не змінюється. Клієнту не повинен у фоновому режимі передавати свою позицію, оскільки вона може не змінюватися, якщо гравець нічого не робить. Достатньо реагувати на дії гравця і передавати саме зміни, наприклад, вектор руху.

По-друге, оптимізація досягається делегуванням певних функцій від сервера до клієнтської частини. Для цього необхідно спершу розділити дані і процеси на критичні та некритичні.

До критичних даних можна віднести положення гравця, його здоров’я та дії, які безпосередньо впливають на ігровий стан та логіку гри: постріл, переміщення чи використання аптечки.

До некритичних можна віднести положення елементів оточення, з якими неможливо взаємодіяти. До прикладу, маленький вазон з квітами, який хоча і має фізичні властивості у самій грі, але не критично, щоб він відображався у однаковій позиції на всіх клієнтах. Отже, серверу не потрібно зберігати дані про його стан. Процес перевірки влучань можна організувати не перевіркою місця влучання фізичної кулі, а перевіркою, що знаходилося у гравця у

прицілі (центр екрану). Таким чином, серверу не треба зберігати дані про кулю і оновлювати її положення у грі, достатньо просто знати, куди у момент пострілу дивився гравець, а анімацію пострілу можна делегувати клієнтській частині. Це логічний хід, оскільки гравець очікує, що він влучить у те, що у прицілі. Також серверу, який відповідає за правомірність подій гри не критично, аби анімація пострілу програвалась однаково у всіх клієнтів. Достатньо того, щоб зниження рівня здоров'я у поціленого гравця було скрізь однакове.

Тобто, сервер передає лише важливі ігрові події, а клієнти самі симулюють фізичні процеси, які відбуваються за тими подіями. Але постає проблема недетермінованості фізики на клієнтському ігровому двигуні. Наприклад, сервер передає клієнтам інформацію про те, що гравець А перемістився на певний вектор. Клієнтські частини можуть по-різному симулювати це переміщення. Це трапляється не часто, але критично важливо для ігрового досвіду, оскільки гравець може бачити одну ціль для пострілу, яка насправді за даними серверу знаходиться в іншому місці. Ця проблема вирішується у спосіб, відповідно до якого сервер передає клієнтам інформацію не тільки зміни, а й про стан всіх об'єктів у світі з певною частотою, яка є значно меншою частоту передачі звичайних даних. Таким чином, досягається певна синхронізація.

1.7 Шаблони програмування

Перед написанням власної реалізації проекту необхідно спланувати архітектуру класів, в тому числі обрати шаблони, за допомогою яких буде відбуватись програмування.

Mediator – шаблон, при якому один екземпляр одного класу агрегує екземпляри іншого класу та підтримує їх комунікацію. При цьому ці екземпляри не знають про один одного і не залежать один від одного. Ідеальний шаблон для агрегування з'єднань між клієнтами та відображеннями гравців в коді.

Command – шаблон, при якому об’єкт відображає інструкцію щодо здійснення певної дії. При цьому цей об’єкт містить всю інформацію для здійснення дії в тому числі об’єкт, над яким здійснюються зміни.

Розділ 2. Реалізація багатокористувацької гри

2.1 Обрані програмні засоби

- GoLang – швидка мова програмування з підтримкою багатопоточності, яку було обрано для написання серверної частини.
- Gorilla/mux – бібліотека для WebSocket-з’єднань.
- Unity – фреймворк для клієнтської частини з вбудованим двигуном фізики та компонентами, які допомагають у розробці.
- C# - мова програмування для скриптів клієнтської частини у Unity.

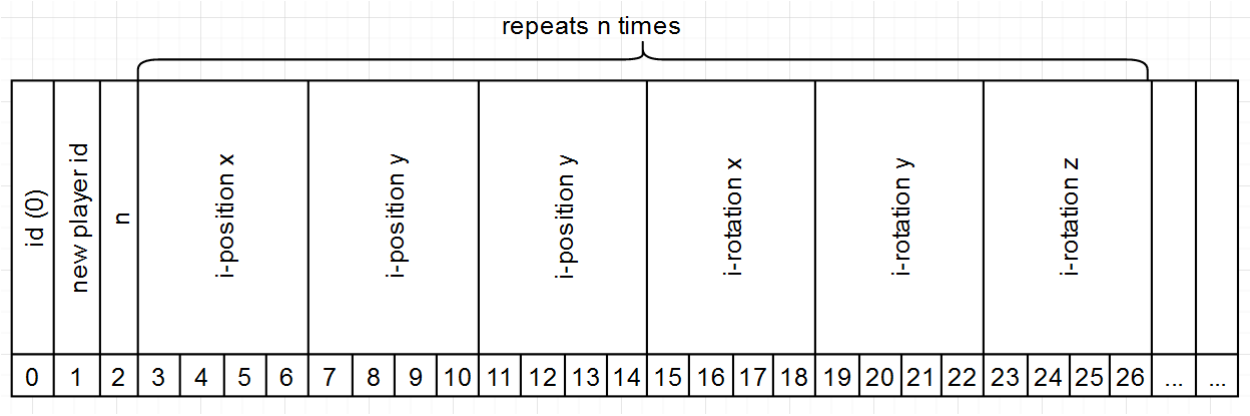
2.2 Пакети передачі даних рівня застосунку

Всі повідомлення мною були розділені на дві групи – **StateMessages**, які ініціюються сервером на надсилають певну інформацію про стан, та **ActionMessages**, які ініціюються клієнтами та є запитами до серверу на зміни даних.

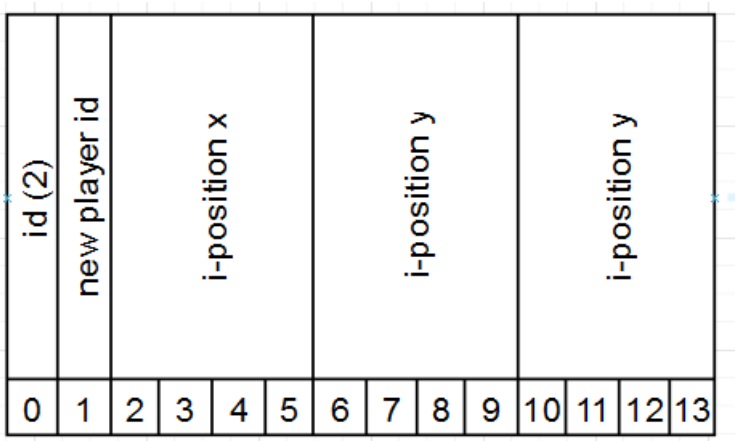
Повідомлення кодуються таким чином, що перший байт - це унікальний номер повідомлення, щоб декодуюча сторона могла правильно декодувати пакет. Далі ідуть байти корисного навантаження, при цьому кожна координата векторів(позиція, переміщення, кут повороту, промені тощо) представлена у 4-байтному числі з плаваючою крапкою, а тому кодується відповідно 4 байтами. Нижче приведені схеми пакетів для повідомлень:

- **InitInfoMessage** надсилається сервером новому гравцю та містить інформацію про згенерований сервером унікальний номер гравця, а також

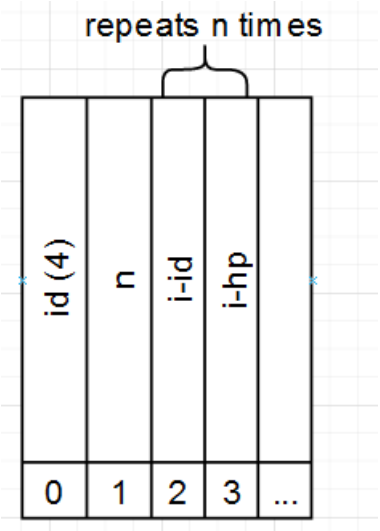
інформація про позицію та кут повороту гравців у ігрових координатах. На одного гравця виділено 24 байти. Для декодування у другому байті написана кількість разів – скільки разів по 24 байти далі в пакеті.



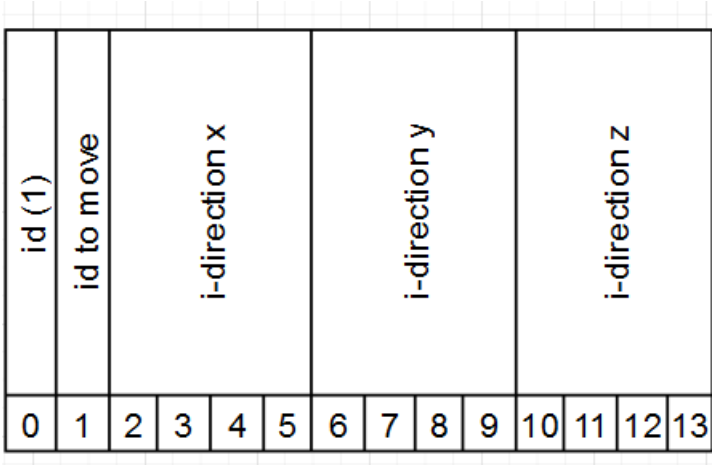
- **PlayerConnectedMessage** надсилається сервером всім гравцям при підключенні нового гравця. Містить унікальний номер цього гравця та позицію для відображення його на клієнтах.



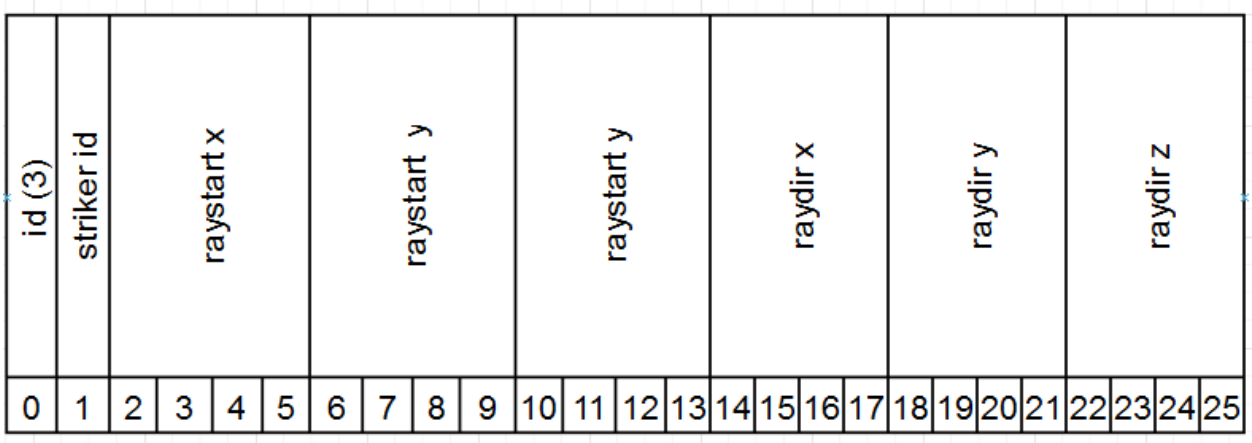
- **HealthDecreaseMessage** надсилається сервером всім гравцям. Містить інформацію, скільки життів у гравця з яким унікальним номером відняти. Для декодування у 1 байті написана кількість разів – скільки разів по 2 байти далі в пакеті.



•**MoveAction** надсилається як сервером, так і клієнтом з інформацією про те, гравець з яким унікальним номером на який вектор перемістився.

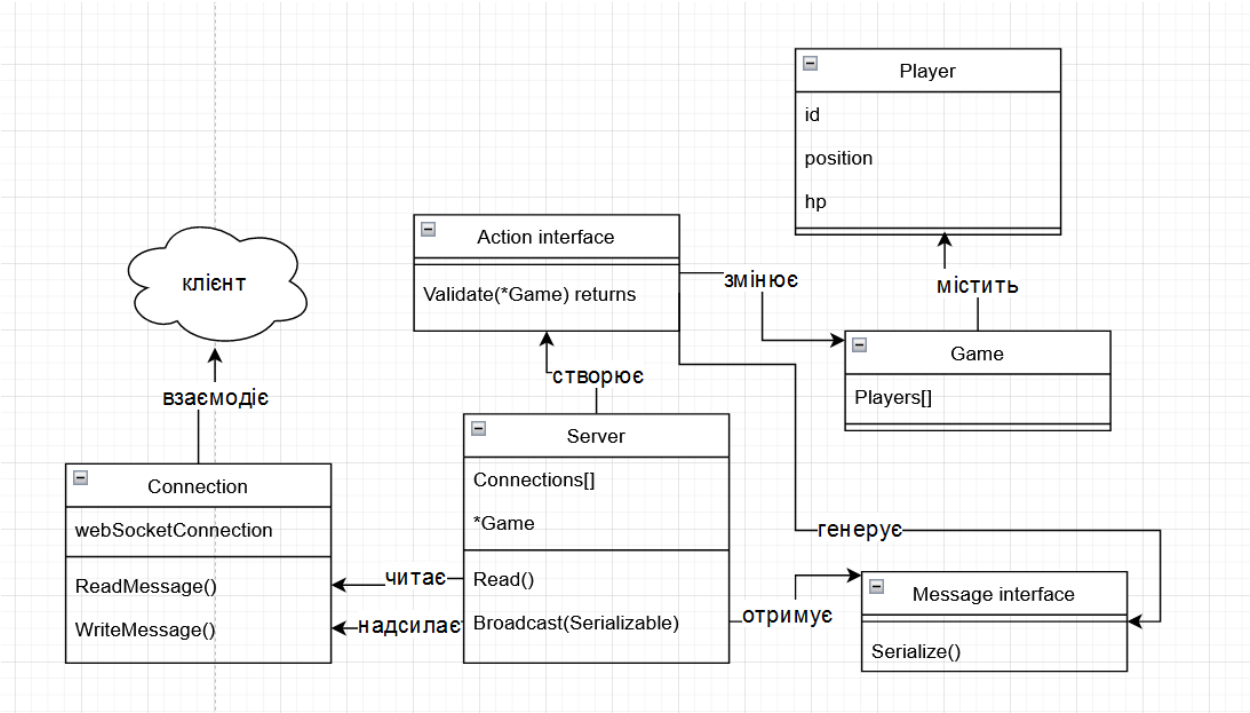


• **FireAction** надсилається клієнтом з інформацією про свій унікальний номер, початковою позицією променя зору та його направленням.



2.3 Розробка серверної частини

Діаграму взаємодії основних класів серверної частини можна побачити нижче:



При розробці серверної частини за основу були взяті такі шаблони програмування, як **Mediator** та **Command**. Клас **Server** є по своїй суті **Mediator** для **Connection**, так само як **Game** для **Player**. **Command** - це є всі реалізації інтерфейсу **Action**, які здійснюють маніпуляції над **Game**. Ці реалізації створюються за допомогою функції-фабрики, що декодує їх з байтового масиву.

Для відображення повідомлень сервера у коді був створений інтерфейс **Message**, реалізації якого серіалізуються у вищезгадані пакети відповідно до протоколу рівня застосунку.

Варто помітити, що з'єднання «слухаються» у окремих потоках, кожен з якого конкурентно доступається до сховища ігрового стану **Game**. Для контролю за конкурентним доступом у мові **GoLang** є клас **mutex.Sync**. Достатньо мати його екземпляр як поле класу **Game** та викликати метод **Lock()** при зчитуванні або зміні даних у класі та **Unlock()** після. Поки не викличеться останній, у всіх інших потоках код зупиниться на методі **Lock()**.

Бібліотека для комунікації по **WebSocket gorilla/mux** не підтримує контроль за конкурентним доступом до сокетів, а тому аналогічні дії проведені над класом **Connection**, що є обгорткою для з'єднань **gorilla/mux**.

Дія **Move** (реалізація **Action**), на жаль, не перевіряється грою на правильність та можливість, але дія **Fire** по заданому від клієнта променю математично перевіряє, чи пересікає він модель якогось гравця відповідно до своїх даних. Буле застосоване рівняння, яке утворилося за допомогою підставлення рівняння променю у рівняння сфери (саме таку форму мають моделі гравців):

$(o + dx – c)^2 = R^2$,

де **o** - вектор початку проміню, **d** – вектор його напряму, **c** – вектор центру сфери, **R** – її радіус, а **x** – точка перетину. Промінь перетинає сферу, якщо це рівняння має розв'язки. Воно має розв'язки, коли дискримінант більше або дорівнює нулю:

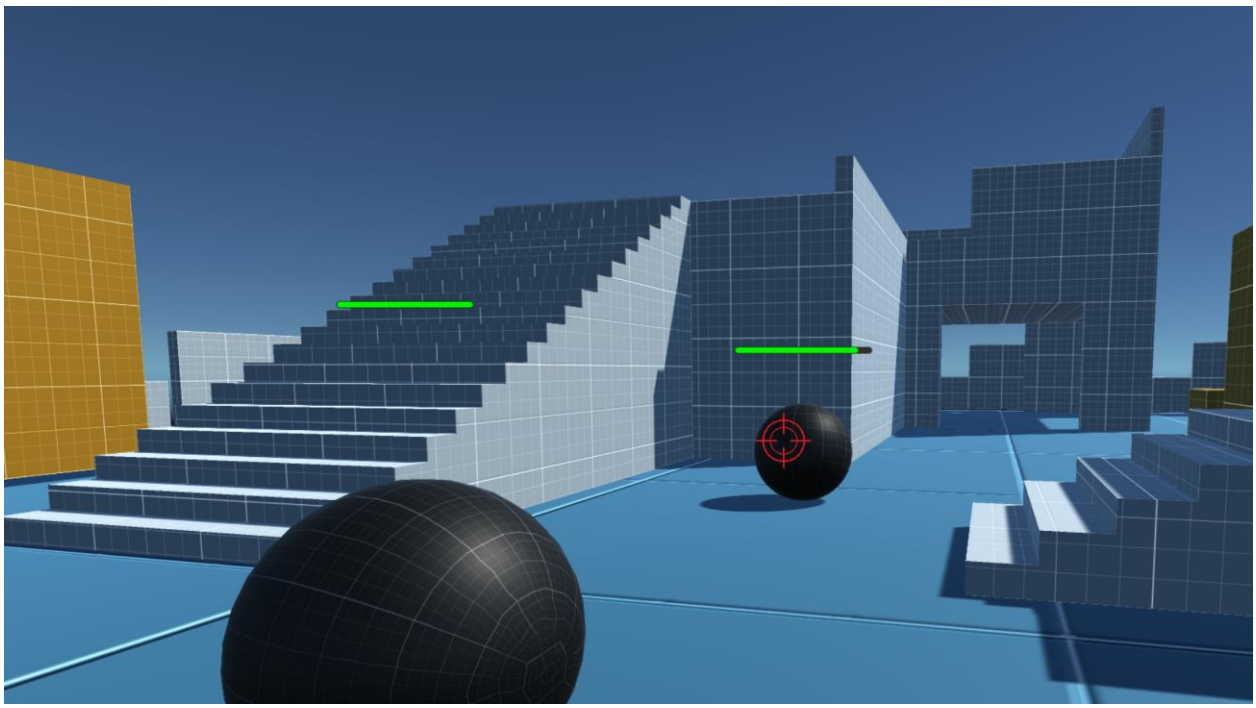
$(d(o-c))^2 – d^2((o-c)^2-R^2) >=0$

На основі цього **Game** генерує список гравців, куди влучив цей промінь та кількість життів, що слід зняти. Сервер передає цю інформацію на клієнти. Більш детальний код можна подивитися у Додатку А.

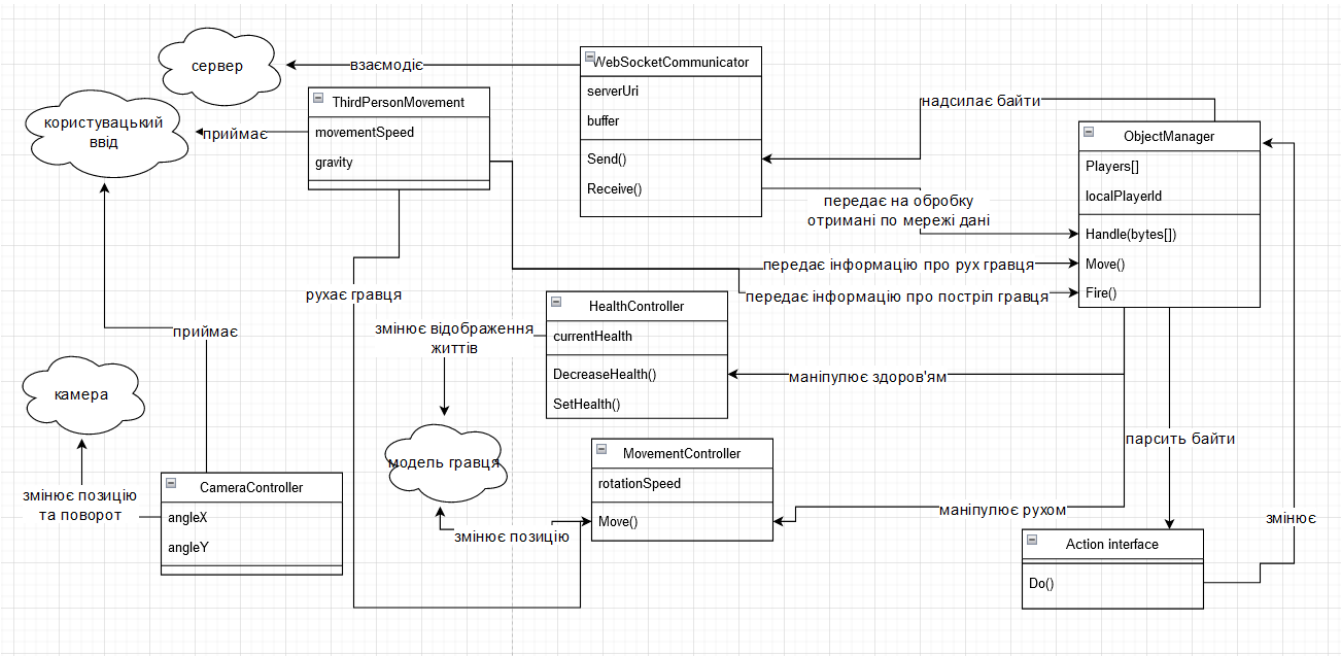
2.4 Розробка клієнтської частини

Розробка клієнтської частини на **Unity** полягає у тому, що б розмістити об'єкти на сцені, та правильно налаштувати їх поведінку за допомогою компонентів та скриптів, які можна прикріпляти до об'єктів.

Знімок екрану з гри можна побачити нижче:



Гравці тут кулі, які управляються за допомогою клавіш “W” “A” “S” “D” та лівою клавішею миші для пострілу. Об’єкти для локального гравця та для симуляції інших гравців на локальній машині були створені окремо, адже вони мають різну логіку поведінки. Фреймворк **Unity** надає багато вбудованого функціоналу, в тому числі симуляцію фізики. Таким чином не було необхідності розробляти та кодувати фізичну модель поведінки, трасування променів тощо. Але для забезпечення функціоналу онлайн-гри, а також управління гравця с пристроїв вводу було написано ряд скриптів. Діаграму їх взаємодії між собою, а також з об’єктами у грі можна побачити нижче:



Вищезгадана різниця між об’єктами локального та віддалених гравців полягає у тому, що тільки до першого прикріплені скрипти слідкування камери **CameraController**, який відслідковує рух миші та змінює положення камери відповідно до гравця та **ThirdPersonMovement**, який реагує на натискання клавіш гравця. На обох об’єктах прикріплені скрипти **MovementController** та **HealthController**, через які за допомогою публічних методів можна маніпулювати об’єктами, рухаючи їх або змінюючи інформацію про кількість життів відповідно.

Скріпт **WebSocketCommunicator** здійснює комунікацію з сервера, приймаючи та надсилаючи повідомлення. Основний «мозок» гри – скрипт **ObjectManager**, який опрацьовує отримані пакети по мережі. Він відповідальний за їх декодування та перетворює отримані пакети у одну з імплементацій інтерфейсу **Action**, який і здійснює відповідні зміни згідно з інструкціями головного серверу. **ObjectManager** містить мапу всіх локальних версій віддалених гравців і за допомогою відкритого API **MovementController** та **HealthController** здійснює маніпуляцію ними. Варто помітити, що **ObjectManager** тільки рухає об’єкти, а за обертання, що є анімацією руху, вони відповідають самі, оскільки це не критично і у сервера нема необхідності передавати це. Більш детальний код скриптів можна подивитися у Додатку Б.

Висновок

Під час написання курсової роботи були досліджені загальні концепції взаємодії клієнта і сервера у розрізі онлайн-гри, поняття конкурентного доступу та оптимізацій ігрового процесу, а також особливості роботи з фреймворком Unity. Були розглянуті переваги та недоліки для ігрової розробки таких протоколів мережевого рівня, як TCP та UDP, а також рівня застосунку.

Також була розроблена реалізація багатокористувацької онлайн-гри з врахуванням підтримки багатьох з’єднань та контролю за правомірністю дій клієнта. Для цього був написаний сервер на мові GoLang, який контролює ігровий світ та відповідає за синхронізацію клієнтів та реагує на їхні дії, а також валідує їх.

Отже, мету досліджень досягнуто.

Перелік використаних джерел

1. Wikipedia [Електронний ресурс]. – URL: https://en.wikipedia.org/wiki/
2. Документація GoLang [Електронний ресурс]. – URL: https://pkg.go.dev/
3. GeeksForGeeks [Електронний ресурс]. – URL: https://www.geeksforgeeks.org/
4. Хабр [Електронний ресурс]. – URL: https://habr.com/
5. Medium [Електронний ресурс]. – URL: https://medium.com/
6. Yalantis [Електронний ресурс]. – URL: https://yalantis.com/
7. Samuel Mortenson [Електронний ресурс]. – URL: https://mortenson.coffee/

Додаток А

Package game:

Game.go

```
package game

import (
	"errors"
	"fmt"
	"sync"
)

type Game struct {
	sync.Mutex
	players map[byte]*Player
}

func (g Game) PlayerIds() []byte {
	var ids []byte
	for id := range g.players {
		ids = append(ids, id)
	}
	return ids
}

func NewGame() *Game {
	return &Game{Mutex: sync.Mutex{}, players: make(map[byte]*Player)}
}
```

```
func (g Game) PlayersTransform(playerId byte) (Transform, bool) {
	player, ok := g.players[playerId]
	if !ok {
		return Transform{}, false
	}
	return player.transform, true
}

func (g Game) Player(playerId byte) (Player, bool) {
	player, ok := g.players[playerId]
	if !ok {
		return Player{}, false
	}
	return *player, true
}

func (g Game) PlayersTransforms() map[byte]Transform {
	transforms := make(map[byte]Transform)
	for i := range g.players {
		transforms[i] = g.players[i].transform
	}
	return transforms
}

func (g Game) Players() map[byte]Player {
	playersCopy := make(map[byte]Player)
	for i := range g.players {
		playersCopy[i] = *g.players[i]
	}
	return playersCopy
```

```
}

func (g *Game) CreatePLayer() (Player, error) {
	g.Lock()
	newId, err := g.generateId()
	if err != nil {
		return Player{}, err
	}
	g.players[newId] = NewPlayer(newId, Vector3{0, 3, 0}, Vector3{0, 0, 0},
100)
	g.Unlock()
	return *g.players[newId], nil
}

func (g Game) generateId() (byte, error) {
	for i := 0; i < MAX_PLAYERS; i++ {
		if _, ok := g.players[byte(i)]; !ok {
			return byte(i), nil
		}
	}
	return 0, errors.New("Max players reached")
}

func (g *Game) MovePlayer(playerId byte, vector Vector3) bool {
	g.Lock()
	player, ok := g.players[playerId]
	if !ok {
		return false
	}
	player.transform.position = AddVectors(player.transform.position, vector)
```

```
	fmt.Printf("changed players position to
%v\n",player.transform.position.String())
	g.Unlock()
	return true
}

func (g *Game) HitPlayer(playerId byte, rayStartPosition, rayDirection Vector3)
(byte, bool) {
	g.Lock()
	defer g.Unlock()
	player, ok := g.players[playerId]
	if !ok {
		return 0, false
	}
	center := player.Transform().Position()
	radius := player.Radius()
	d := SubtractVectors(rayStartPosition, center)
	b := 2 * ScalarMultiplication(rayDirection, d)
	a := ScalarMultiplication(rayDirection, rayDirection)
	c := ScalarMultiplication(d, d) - radius*radius
	//D :=
4*float32(math.Pow(float64(game.ScalarMultiplication(f.rayDirection,
game.SubtractVectors(f.rayStartPosition, center))), 2)) -
4*game.ScalarMultiplication(f.rayDirection,
f.rayDirection)*(game.ScalarMultiplication(game.SubtractVectors(f.rayStartPositi
on, center), game.SubtractVectors(f.rayStartPosition, center))-radius*radius)
	D := b*b - 4*a*c
	fmt.Println(D)
	if D >= 0 {
		player.hp -= DAMAGE
```

```
            if player.hp < 0 {
                res:= DAMAGE + player.hp
                delete(g.players, playerId)
                return res, true
            }
            return DAMAGE, true
        }
        return 0, false
}
```

Types.go

```
package game

const RADUIS float32 = 0.5
const DAMAGE byte = 10

const MAX_PLAYERS = 10



type Transform struct {
        position, rotation Vector3
}

func (t Transform) Position() Vector3 {
        return t.position
}

func NewTransform(position Vector3, rotation Vector3) *Transform {
        return &Transform{position: position, rotation: rotation}
}
```

```
// 18 bytes
func (t Transform) Serialize() []byte {
	data := append(t.position.Serialize(), t.rotation.Serialize()...)
	return data
}

type Player struct {
	id	byte
	transform Transform
	hp	byte
}

func (p *Player) Id() byte {
	return p.id
}

func (p *Player) Transform() Transform {
	return p.transform
}

func (p *Player) Hp() byte {
	return p.hp
}

func (p *Player) Radius() float32 {
	return RADUIS
}

func NewPlayer(id byte, position Vector3, rotation Vector3, hp byte) *Player {
```

```
	return &Player{id: id, transform: Transform{position: position, rotation:
rotation}, hp: hp}
}
```

Vector.go

```
package game

import (
	"encoding/binary"
	"fmt"
	"math"
)

type Vector3 struct {
	x, y, z float32
}

func NewVector3(x float32, y float32, z float32) *Vector3 {
	return &Vector3{x: x, y: y, z: z}
}

func AddVectors(v1, v2 Vector3) Vector3 {
	return Vector3{x: v1.x + v2.x, y: v1.y + v2.y, z: v1.z + v2.z}
}

func SubtractVectors(v1, v2 Vector3) Vector3 {
	return Vector3{x: v1.x - v2.x, y: v1.y - v2.y, z: v1.z - v2.z}
}

func ScalarMultiplication(v1,v2 Vector3) float32{
```

```
	return v1.x*v2.x+v1.y*v2.y+v1.z*v2.z
}

func Multiplicate(n float32,v1 Vector3) Vector3{
	return  Vector3{x: v1.x*n, y: v1.y*n, z: v1.z*n}
}

// 9 bytes
func (v Vector3) Serialize() []byte {
	data := [12]byte{}
	binary.LittleEndian.PutUint32(data[0:4], math.Float32bits(v.x))
	binary.LittleEndian.PutUint32(data[4:8], math.Float32bits(v.y))
	binary.LittleEndian.PutUint32(data[8:], math.Float32bits(v.z))
	return data[:]
}

func (v Vector3) String() string {
	return fmt.Sprintf("%v %v %v", v.x, v.y, v.z)
}
```

Package networking:

Actions.go

```
package networking

import (
	"encoding/binary"
	"errors"
	"fmt"
	"github.com/aabuddabi/unitygame/internal/game"
	"math"
```

```
)

const (
	MOVE_PKG_LENGH = 14
	FIRE_PKG_LENGH = 26
)

type MoveAction struct {
	moverId	byte
	positionDelta game.Vector3
}

func (m MoveAction) String() string {
	return fmt.Sprintf("player #%v moved on vector [%s]", m.moverId,
m.positionDelta.String())
}

func (m MoveAction) Serialize() []byte {
	fmt.Printf("sending pos delta: %v\n",m.positionDelta.String())
	return append([]byte{1, m.moverId}, m.positionDelta.Serialize()...)
}

func (m MoveAction) Validate(game *game.Game) (StateMessage, bool) {
	ok := game.MovePlayer(m.moverId, m.positionDelta)
	if !ok {
		return nil, false
	}
	return m, true
}
```

```
func NewAction(initializerId byte, data []byte) (Action, error) {
	if len(data) == 0 {
		return nil, errors.New("wrong length of package")
	}
	actionType := data[0]
	switch actionType {
	case 1:
		//fmt.Printf("player #%v sent message %v\n", initializerId, data)
		if len(data) != MOVE_PKG_LENGH {
			return nil, errors.New("wrong length of MoveAction package")
		}
		if data[1] != initializerId {
			return nil, errors.New(fmt.Sprintf("Connection id %v differs
from moved player %v!", initializerId, data[1]))
		}
		x := math.Float32frombits(binary.LittleEndian.Uint32(data[2:6]))
		y := math.Float32frombits(binary.LittleEndian.Uint32(data[6:10]))
		z := math.Float32frombits(binary.LittleEndian.Uint32(data[10:]))
		fmt.Printf("got movement y %v ", y)
		return &MoveAction{moverId: initializerId, positionDelta:
*game.NewVector3(x, y, z)}, nil
	case 3:
		if len(data) != FIRE_PKG_LENGH {
			return nil, errors.New("wrong length of MoveAction package")
		}
		if data[1] != initializerId {
			return nil, errors.New("Connection id differs from moved
player!")
		}
```

```
		originX :=
math.Float32frombits(binary.LittleEndian.Uint32(data[2:6]))
		originY :=
math.Float32frombits(binary.LittleEndian.Uint32(data[6:10]))
		originZ :=
math.Float32frombits(binary.LittleEndian.Uint32(data[10:14]))
		directionX :=
math.Float32frombits(binary.LittleEndian.Uint32(data[14:18]))
		directionY :=
math.Float32frombits(binary.LittleEndian.Uint32(data[18:22]))
		directionZ :=
math.Float32frombits(binary.LittleEndian.Uint32(data[22:]))
		return &FireAction{strikerId: initializerId, rayStartPosition:
*game.NewVector3(originX, originY, originZ), rayDirection:
*game.NewVector3(directionX, directionY, directionZ)}, nil
	default:
		return nil, fmt.Errorf("unexpected action type %v", actionType)
	}
}

type FireAction struct {
	strikerId		byte
	rayStartPosition, rayDirection game.Vector3
}

func NewFireAction(strikerId byte, rayStartPosition game.Vector3, rayDirection
game.Vector3) *FireAction {
	return &FireAction{strikerId: strikerId, rayStartPosition: rayStartPosition,
rayDirection: rayDirection}
}
```

```
func (f FireAction) Validate(g *game.Game) (StateMessage, bool) {
	var ids, hps []byte
	playerIds := g.PlayerIds()
	for playerId := range playerIds {
		player, ok := g.Player(byte(playerId))
		if !ok {
			panic("")
		}
		if byte(playerId) != f.strikerId {
			hpDelta, ok := g.HitPlayer(player.Id(), f.rayStartPosition,
f.rayDirection)
			if !ok {
				break
			}
			fmt.Printf("hit player #%v with %v\n", player.Id(), hpDelta)
			ids = append(ids, byte(playerId))
			hps = append(hps, hpDelta)
			fmt.Printf("%v | %v\n", ids, hps)
		}
	}
	return NewHealthDecreaseMessage(ids, hps), true
}

func (f FireAction) String() string {
	return fmt.Sprintf("this is fire action")
}
```

Server.go

```
package networking
```

```
import (
	"errors"
	"fmt"
	"github.com/aabuddabi/unitygame/internal/game"
	"github.com/gorilla/websocket"
	"net/http"
	"sync"
)

const MAX_CONNS byte = 10

var upgrader = websocket.Upgrader{
	CheckOrigin: func(r *http.Request) bool {
		return true
	},
}

type Connection struct {
	sync.Mutex
	websocket.Conn
	playerId byte
}

func NewConnection(conn websocket.Conn, playerId byte) *Connection {
	return &Connection{Mutex: sync.Mutex{}, Conn: conn, playerId: playerId}
}

type Server struct {
	game	*game.Game
```

```
	connections   map[byte]*Connection
	handleMessage func(message []byte)
}

func StartServer(handleMessage func(message []byte), game *game.Game) error {
	server := Server{
		game,
		make(map[byte]*Connection),
		handleMessage,
	}

	http.HandleFunc("/", server.handleNewConnection)
	fmt.Println("Listening")
	err := http.ListenAndServe("localhost:8080", nil)
	if err != nil {
		fmt.Println("Failed to start")
		return err
	}
	return nil
}

func (s *Server) handleNewConnection(w http.ResponseWriter, r *http.Request) {
	fmt.Println("Incoming connection...")
	connection, _ := upgrader.Upgrade(w, r, nil)
	fmt.Println("Upgraded to WS protocol")

	newConnectionId, err := s.generateId()
	if err != nil {
		w.Write([]byte(fmt.Sprintf("500 - %s", err.Error())))
	}
```

```
	newPlayer, err := s.game.CreatePLayer()
	if err != nil {
		w.Write([]byte(fmt.Sprintf("500 - %s", err.Error())))
	}
	s.connections[newConnectionId] = NewConnection(*connection,
newPlayer.Id())

	msg := NewInitInfoMessage(newPlayer.Id(), s.game.PlayersTransforms())
	connection.WriteMessage(websocket.BinaryMessage, msg.Serialize())

	newcomerInfoMsg := NewPlayerConnectedMessage(newPlayer.Id(),
newPlayer.Transform().Position())
	s.Broadcast(newcomerInfoMsg.Serialize())

	go func(playerId byte) {
		fmt.Printf("starting gorourine with id #%v\n", playerId)
		defer connection.Close()
		defer delete(s.connections, newConnectionId)
		for {
			mt, data, err := connection.ReadMessage()

			if err != nil || mt == websocket.CloseMessage {
				fmt.Printf("Connection closed with error %s", err.Error())
				break
			}

			action, err := NewAction(playerId, data)
			if err != nil {
				fmt.Printf("Connection closed with error %s", err.Error())
				break
```

```
                        }
                        fmt.Println(action.String())
                        msg, ok := action.Validate(s.game)
                        if ok {
                                s.Broadcast(msg.Serialize())
                        }
                }
        }(newPlayer.Id())
}

func (s *Server) Broadcast(data []byte) {
        //fmt.Printf("player #%v broadcasting message %v\n", data)
        for i := range s.connections {
                s.connections[i].Lock()
                s.connections[i].WriteMessage(websocket.BinaryMessage, data)
                s.connections[i].Unlock()
        }
}

func (s Server) generateId() (byte, error) {
        for i := 0; i < int(MAX_CONNS); i++ {
                if _, ok := s.connections[byte(i)]; !ok {
                        return byte(i), nil
                }
        }
        return 0, errors.New("Max connections reached")
}
```

State.go

```
package networking
```

```
import (
	"github.com/aabuddabi/unitygame/internal/game"
)

type InitInfoMessage struct {
	id         byte
	transforms map[byte]game.Transform
}

func NewInitInfoMessage(id byte, transforms map[byte]game.Transform)
*InitInfoMessage {
	return &InitInfoMessage{id: id, transforms: transforms}
}

func (i InitInfoMessage) Serialize() []byte {
	data := []byte{0, i.id, byte(len(i.transforms))}
	for id := range i.transforms {
		data = append(data, id)
		data = append(data, i.transforms[id].Serialize()...)
	}
	return data
}

type PlayerConnectedMessage struct {
	newcomerId byte
	position   game.Vector3
}
```

```
func NewPlayerConnectedMessage(id byte, position game.Vector3)
*PlayerConnectedMessage {
	return &PlayerConnectedMessage{newcomerId: id, position: position}
}

func (pc PlayerConnectedMessage) Serialize() []byte {
	data := []byte{2, pc.newcomerId}
	data = append(data, pc.position.Serialize()...)
	return data
}

type HealthDecreaseMessage struct {
	ids []byte
	hps []byte
}

func NewHealthDecreaseMessage(ids []byte, hps []byte) *HealthDecreaseMessage
{
	return &HealthDecreaseMessage{ids: ids, hps: hps}
}

func (h HealthDecreaseMessage) Serialize() []byte {
	idLength:=len(h.ids)
	data:= make([]byte,idLength*2+2)
	data[0]=4
	data[1]=byte(idLength)
	for i := 0; i < idLength; i++ {
		data[2+i*2] = h.ids[i]
		data[2+i*2+1] = h.hps[i]
	}
```

```
	return data
}
```

Types.go

```
package networking

import (
	"fmt"
	"github.com/aabuddabi/unitygame/internal/game"
)

type Serializer interface {
	Serialize() []byte
}
type StateMessage interface {
	Serializer
}

type Action interface {
	Validate(*game.Game) (StateMessage,bool)
	fmt.Stringer
	//Undo()
}
```

Package main:

Main.go

```
package main

import (
	"fmt"
```

```
        "github.com/aabuddabi/unitygame/internal/game"
        "github.com/aabuddabi/unitygame/internal/networking"
)

func main() {

        game:=game.NewGame()

        networking.StartServer(messageHandler,game)
}

func messageHandler(message []byte) {
        fmt.Println(string(message))
}
```

Додаток В

CameraController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraController : MonoBehaviour
{
    public float rotationSpeed;
    public Transform target;

    private float angleY;
    private float angleX;

    void Start()
```

```
    {
        angleY = transform.rotation.y;
        angleX = transform.rotation.x;
    }

    // Update is called once per frame
    void Update()
    {
        angleY += rotationSpeed * Input.GetAxis("Mouse X");
        angleX -= rotationSpeed * Input.GetAxis("Mouse Y");

        transform.position = target.transform.position;
        transform.rotation = Quaternion.Euler(angleX, angleY, 0);
    }
}
```

HealthController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class HealthController : MonoBehaviour
{
    public GameObject hbPrefab;
    private GameObject hb;
    public byte initialHealth;
    private byte currentHealth;
    // Start is called before the first frame update
    void Start()
```

```
    {
        hb = Instantiate(hbPrefab, GameObject.Find("HealthCanvas").transform);
        currentHealth = initialHealth;
        hb.GetComponentInChildren<Slider>().value = currentHealth;
    }

    // Update is called once per frame
    void Update()
    {
        hb.transform.position =
Camera.main.WorldToScreenPoint(transform.position+ new Vector3(0,1,0));
    }

    public void SetHealth(byte newHealth)
    {
        currentHealth = newHealth;
        hb.GetComponentInChildren<Slider>().value = newHealth;
    }

    public void DecreaseHealth(byte deltaHealth)
    {
        currentHealth -= deltaHealth;
        hb.GetComponentInChildren<Slider>().value = currentHealth;
    }
}
```

MovementController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class MovementController : MonoBehaviour
{

    private ObjectManager objManager;
    private CharacterController controller;
    public float rotationSpeed;
    public float gravity;

    void Start()
    {
        controller = GetComponent<CharacterController>();
        objManager = GetComponent<ObjectManager>();
    }

    public void Move(Vector3 movementDirection)
    {
        Vector3 rotationAngles = Vector3.Cross(Vector3.up, movementDirection) *
60;
        transform.Rotate(rotationAngles, Space.World);
        //Debug.Log("I'm moving on vector: " + movementDirection);
        controller.Move(movementDirection);
    }
    public void SetPosition(Vector3 newPosition)
    {
        controller.Move(newPosition - transform.position);
    }
}
```

ObjectManager.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ObjectManager : MonoBehaviour
{
    private byte localPlayerId;
    public GameObject remotePlayer;
    public WebSocketCommunicator wsc;
    public Dictionary<byte, GameObject> players = new Dictionary<byte,
GameObject>();

    // Start is called before the first frame update
    void Start()
    {
        wsc = GetComponent<WebSocketCommunicator>();
    }

    public GameObject GetPlayer(byte id)
    {
        return players[id];
    }

    public void Move(Vector3 vector)
    {
        MoveAction moveAction = new MoveAction(this, localPlayerId, vector);
        //Debug.Log("Got moveAction from player: " + localPlayerId);
        wsc.Send(moveAction.Serialize());
    }
```

```
    public void Fire(Vector3 origin, Vector3 direction)
    {
        FireMessage fireMessage = new FireMessage(this, localPlayerId, origin,
direction);
        //Debug.Log("Got fireAction from player: " + fireMessage.ToString());
        wsc.Send(fireMessage.Serialize());
    }

    public void Handle(ArraySegment<byte> b)
    {
        Action action = CreateActionFromBytes(b);
        action.Do();
    }



    public Action CreateActionFromBytes(ArraySegment<byte> buf)
    {
        switch (buf.Array[0])
        {
            case 0:
                byte initId;
                byte[] initIds;
                Vector3[,] transforms;
                transforms = new Vector3[buf.Array[2], 2];
                initIds = new byte[buf.Array[2]];
                initId = buf.Array[1];
                //Debug.Log("initial id: " + initId);
                for (int i = 0; i < buf.Array[2]; i++)
```

```
            {
                initIds[i] = buf.Array[25 * i + 3];
                transforms[i, 0].x = BitConverter.ToSingle(buf.Array, 25 * i + 4);
                transforms[i, 0].y = BitConverter.ToSingle(buf.Array, 25 * i + 8);
                transforms[i, 0].z = BitConverter.ToSingle(buf.Array, 25 * i + 12);
                transforms[i, 1].x = BitConverter.ToSingle(buf.Array, 25 * i + 16);
                transforms[i, 1].y = BitConverter.ToSingle(buf.Array, 25 * i + 20);
                transforms[i, 1].z = BitConverter.ToSingle(buf.Array, 25 * i + 24);

            }
            InitAction initAction = new InitAction(this, initId, initIds, transforms);
            return initAction;
        case 1:
            byte idToMove = buf.Array[1];
            float moveVectorX = BitConverter.ToSingle(buf.Array, 2);
            float moveVectorY = BitConverter.ToSingle(buf.Array, 6);
            float moveVectorZ = BitConverter.ToSingle(buf.Array, 10);
            MoveAction moveAction = new MoveAction(this, idToMove, new
Vector3(moveVectorX, moveVectorY, moveVectorZ));
            return moveAction;
        case 2:
            byte idToInstantiate = buf.Array[1];
            float posVectorX = BitConverter.ToSingle(buf.Array, 2);
            float posVectorY = BitConverter.ToSingle(buf.Array, 6);
            float posVectorZ = BitConverter.ToSingle(buf.Array, 10);
            AddPlayerAction addPlayerAction = new AddPlayerAction(this,
idToInstantiate, new Vector3(posVectorX, posVectorY, posVectorZ));
            return addPlayerAction;
        case 4:
```

```
            HealthDecreaseAction healthDecreaseAction = new
HealthDecreaseAction(this);
            int arraysLength = buf.Array[1];
            byte[] ids = new byte[arraysLength];
            byte[] hps = new byte[arraysLength];
            for (int i = 0; i < arraysLength; i++)
            {
               ids[i] = buf.Array[2+i*2];
               hps[i] = buf.Array[2 + i * 2 + 1];
            }
            healthDecreaseAction.ids = ids;
            healthDecreaseAction.hps = hps;
            return healthDecreaseAction;
         default:
            return null;
      }
   }

   public interface Action
   {
      void Do();
      string ToString();
   }

   public interface Message
   {
      ArraySegment<byte> Serialize();
   }

   public class InitAction : Action
```

```
    {
        public byte id;
        public byte[] ids;
        public Vector3[,] transforms;
        private ObjectManager objectManager;

        public InitAction(ObjectManager objectManager,byte id,byte[] ids, Vector3[,]
transforms)
        {
            this.objectManager = objectManager;
            this.id = id;
            this.ids = ids;
            this.transforms = transforms;
        }

        public void Do()
        {
            objectManager.localPlayerId = id;
            //Debug.Log("set local player id: " + objectManager.localPlayerId);
            for (int i = 0; i < ids.Length; i++)
            {
                Vector3 newPosition = transforms[i,0];
                Quaternion newRotation = Quaternion.Euler(transforms[i,1]);
                GameObject newPlayer;
                if (i == id)
                {

objectManager.GetComponent<MovementController>().SetPosition(newPosition);
                    objectManager.transform.rotation = newRotation;
                    objectManager.players[ids[i]] = objectManager.gameObject;
```

```
                break;
            }
            newPlayer = Instantiate(objectManager.remotePlayer, newPosition,
newRotation);
            objectManager.players[ids[i]] = newPlayer;
        }
        objectManager.GetComponent<ThirdPersonMovement>().enabled = true;
        objectManager.GetComponent<HealthController>().enabled = true;
        //objectManager.GetComponent<MovementController>().enabled = true;
    }
    public string ToString()
    {
        return "this is init message";
    }
}

public class MoveAction : Action,Message
{
    public byte id;
    public Vector3 vector;
    private ObjectManager objectManager;

    public MoveAction(ObjectManager objectManager, byte id, Vector3 vector)
    {
        this.objectManager = objectManager;
        this.id = id;
        this.vector = vector;
    }

    public void Do()
```

```
        {
            if (objectManager.localPlayerId != id)
            {

objectManager.GetPlayer(id).GetComponent<MovementController>().Move(vecto
r);
            }
        }

        public ArraySegment<byte> Serialize()
        {
            byte[] data = new byte[14];
            data[0] = 1;
            //Debug.Log("Serializing localplayer id: " + data[1]);
            data[1] = id;
            BitConverter.GetBytes(vector.x).CopyTo(data,2);
            BitConverter.GetBytes(vector.y).CopyTo(data, 6);
            BitConverter.GetBytes(vector.z).CopyTo(data, 10);
            return new ArraySegment<byte>(data);
        }
        public string ToString()
        {
            return "Move on Vector "+vector.ToString()+" #"+id;
        }
    }

    public class AddPlayerAction : Action
    {
        public byte id;
        public Vector3 position;
```

```
        private ObjectManager objectManager;

        public AddPlayerAction(ObjectManager objectManager, byte id, Vector3
position)
        {
            this.objectManager = objectManager;
            this.id = id;
            this.position = position;
        }

        public void Do()
        {
            if (objectManager.localPlayerId != id)
            {
                GameObject newPlayer = Instantiate(objectManager.remotePlayer,
position, Quaternion.identity);
                objectManager.players[id] = newPlayer;
            }
        }



        public string ToString()
        {
            return "NewPlayer at Position "+ position.ToString()+" #"+id;
        }
    }

    public class FireMessage : Message
    {
        public byte id;
```

```
    public Vector3 origin;
    public Vector3 direction;
    private ObjectManager objectManager;

    public FireMessage(ObjectManager objectManager, byte id, Vector3 origin,
Vector3 direction)
    {
        this.objectManager = objectManager;
        this.id = id;
        this.direction = direction;
        this.origin = origin;
    }

    public ArraySegment<byte> Serialize()
    {
        byte[] data = new byte[26];
        data[0] = 3;
        data[1] = id;
        BitConverter.GetBytes(origin.x).CopyTo(data, 2);
        BitConverter.GetBytes(origin.y).CopyTo(data, 6);
        BitConverter.GetBytes(origin.z).CopyTo(data, 10);
        BitConverter.GetBytes(direction.x).CopyTo(data, 14);
        BitConverter.GetBytes(direction.y).CopyTo(data, 18);
        BitConverter.GetBytes(direction.z).CopyTo(data, 22);
        return new ArraySegment<byte>(data);
    }
    public string ToString()
    {
        return "Fire in direction " + direction.ToString() + " in position " +
origin.ToString() + " #" + id;
```

```
        }
    }

    public class HealthDecreaseAction : Action
    {
        public byte[] ids;
        public byte[] hps;
        private ObjectManager objectManager;

        public HealthDecreaseAction(ObjectManager objectManager)
        {
            this.objectManager = objectManager;
        }

        public void Do()
        {
            for (int i = 0; i < ids.Length; i++)
            {

objectManager.GetPlayer(ids[i]).GetComponent<HealthController>().DecreaseHe
alth(hps[i]);
            }
        }
        public string ToString()
        {
            return "this is init message";
        }
    }
}
```

ThirdPersonMovement.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ThirdPersonMovement : MonoBehaviour
{

    public Transform camera;



    public float gravity;

    public float movementSpeed;
    private CharacterController controller;
    private ObjectManager objManager;



    // Start is called before the first frame update
    void Start()
    {
        controller = GetComponent<CharacterController>();
        objManager = GetComponent<ObjectManager>();
    }

    // Update is called once per frame
    void Update()
    {
        Move();
        Fire();
```

```
    }

    private void Move()
    {
        float sideComponent = Input.GetAxis("Horizontal");
        float forwardComponent = Input.GetAxis("Vertical");

        Vector3 movementDirection = (sideComponent * camera.right +
forwardComponent * camera.forward).normalized;
        movementDirection *= movementSpeed * Time.deltaTime;
        movementDirection.y -= gravity * Time.deltaTime;

        Vector3 prevPos = transform.position;
        GetComponent<MovementController>().Move(movementDirection);
        Vector3 actualPos = transform.position;
        //Debug.Log("moving " + movementDirection.y);
        if (actualPos - prevPos != Vector3.zero)
        {
            Debug.Log("moving " + movementDirection.x + " " +
movementDirection.y + " " + movementDirection.z);
            objManager.Move(actualPos-prevPos);
        }
    }

    private void Fire()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Ray shootRay = Camera.main.ViewportPointToRay(new Vector3(0.5F,
0.5F, 0));
```

```
            objManager.Fire(shootRay.origin,shootRay.direction);
        }
    }
}
```

WebSocketCommunicator.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Text;
using System.Threading;
using System.Net.WebSockets;
using UnityEngine;

public class WebSocketCommunicator : MonoBehaviour
{
    private ObjectManager objManager;

    Uri u = new Uri("ws://localhost:8080");
    ClientWebSocket cws = null;
    ArraySegment<byte> buf = new ArraySegment<byte>(new byte[1024]);

    void Start() {
        objManager = GetComponent<ObjectManager>();
        Connect();
    }

    async void Connect()
    {
```

```
        cws = new ClientWebSocket();
        try
        {
            await cws.ConnectAsync(u, CancellationToken.None);
            if (cws.State == WebSocketState.Open) Debug.Log("connected");
            Receive();
        }
        catch (Exception e) { Debug.Log("woe " + e.Message); }
    }

    public async void Send(ArraySegment<byte> b)
    {
        //Debug.Log("sending bytes: " + b.Array);
        await cws.SendAsync(b, WebSocketMessageType.Text, true,
CancellationToken.None);
    }

    async void Receive()
    {
        WebSocketReceiveResult r = await cws.ReceiveAsync(buf,
CancellationToken.None);
        //Debug.Log("received bytes: " + buf.Array);
        objManager.Handle(buf);
        Receive();
    }
}
```