Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій

УДК 004.75;004.042

На правах рукопису

ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Тема: “Розробка відеогри за мотивами

“World Chase Tag””

Спеціальність – 121 “Інженерія програмного забезпечення”

ПОЯСНЮВАЛЬНА ЗАПИСКА

БР.ІПЗ – 30.00.00.000

Студент
ІПЗ-41 _____ /Артур КАШУБА/

Науковий керівник
д.т.н. с.н.с. _____ /Геннадій ПОРЄВ/

Консультант
з питань нормоконтролю
фахівець _____ /Тамара ЧАПОВСЬКА/

Допускається до захисту
Завідувач кафедри
д.т.н., проф. _____ /Олексій БИЧКОВ/

Київ – 2021

Рішенням Екзаменаційної комісії
випускна кваліфікаційна робота студента

_____

захищена з оцінкою

_____

Голова Екзаменаційної комісії

професор, доктор техн. наук Вишнівський Віктор Вікторович

Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій
Спеціальність 6.050103 Програмна інженерія

ЗАТВЕРДЖЕНО
Завідувач кафедри
програмних систем і технологій
_____(Олексій БИЧКОВ)
„___”_____ 2021р.

ЗАВДАННЯ
НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ
СТУДЕНТУ

Кашубі Артуру Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема випускної кваліфікаційної бакалаврської роботи «Розробка відеогри за мотивами “World Chase Tag”»
затверджена наказом вищого навчального закладу від „__”_____20__р. №_____
2. Строк здачі студентом закінченої роботи _____
3. Вихідні дані до роботи _____
_____
_____
4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)
_____
_____
_____
5. Перелік графічного матеріалу (з точним забезпеченням обов’язкових креслень)
_____
_____
_____
_____
_____

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 13 жовтня 2020 р.

Керівник _____ (Геннадій ПОРЄВ)

Завдання прийняв до виконання ____ _____ (Артур КАШУБА)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Збір інформації	24.10.2020	
2	Вивчення варіантів реалізації та вибір варіанту для розробки	15.01.2021	
3	Розробка графічної складової гри	03.02.2021	
4	Вибір паттерну	19.02.2021	
5	Проектування архітектури	12.03.2021	
6	Реалізація зв’язку сервер-клієнт	28.03.2021	
7	Аналіз готового продукту та оформлення дипломної роботи	25.04.2021	

Студент – бакалавр _____(Артур КАШУБА)

Керівник роботи _____(Геннадій ПОРЄВ)

АНОТАЦІЯ

Випускна кваліфікаційна бакалаврська робота: 68 с., 22 рис., 8 табл., 3 додат., 33 джерела.

Тема: Розробка відеогри за мотивами “World Chase Tag”

Мета роботи: Створення архітектури побудованої на Unity-components system для комп’ютерної гри.

Об’єкт дослідження: процес розробки архітектури для створення комп’ютерних ігор

Предмет дослідження: Об’єктно-орієнтована парадигма програмування, компонентно-орієнтований підхід.

Результат

Досліджено можливість застосування компонентно-орієнтований підходу, що забезпечує створення і повторне використання компонентів. Завершена реалізація компонентно-орієнтований підходу, що представляє собою комп’ютерну гру.

Висновки

Був створений проект “CatchUp” за мотивами міжнародного чемпіонату “World Chase Tag”. На даний момент проект є цілком готовий до виходу на ринок.

Вирішені проблеми пов’язані з передачею повідомлень на сервер та їх оптимізацією. Завдяки цьому “CatchUp” не потребує великих обчислювальних потужностей на стороні сервера та пропускної здатності на клієнта (B/s).

АННОТАЦИЯ

Выпускная квалификационная бакалаврская работа: 68 с., 22 рис., 8 табл., 3 доп., 33 источника.

Тема: Разработка видеоигры по мотивам "World Chase Tag"

Цель работы: Создание архитектуры построенной на Unity-components system для компьютерной игры.

Объект исследования: процесс разработки архитектуры для создания компьютерных игр

Предмет исследования: объектно-ориентированная парадигма программирования, компонентно-ориентированный подход.

Результат

Исследована возможность применения компонентно-ориентированный подход, обеспечивающий создание и повторное использование компонентов. Завершена реализация компонентно-ориентированный подход, представляющий собой компьютерную игру.

Выводы

Был создан проект "CatchUp" по мотивам международного чемпионата "World Chase Tag". На данный момент проект полностью готов к выходу на рынок.

Решены проблемы, связанные с передачей сообщений на сервер и их оптимизацией. Благодаря этому "CatchUp" не требует больших вычислительных мощностей на стороне сервера и пропускной способности на клиента (B/s).

SUMMARY

Final qualifying bachelor's thesis: 68 pages, 22 figures, 8 tables, 3 appendices, 33 sources.

Topic: Development of a video game based on "World Chase Tag"

Objective: To create an architecture built on the Unity-components system for a computer game.

Object of research: the process of developing an architecture for creating computer games

Subject of research: Object-oriented programming paradigm, component-oriented approach.

Result

The possibility of applying a component-oriented approach that provides the creation and reuse of components has been investigated. Completed the implementation of a component-oriented approach, which is a computer game.

Conclusions

The project "CatchUp" was created based on the international championship "World Chase Tag". At the moment, the project is quite ready to enter the market.

Resolved issues related to sending messages to the server and optimizing them. Due to this, "CatchUp" does not require large computing power on the server side and bandwidth per client (B / s).

ЗМІСТ

ПЕРЕЛІК ОСНОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ

UnityScript – важливий компонент усіх програм в Unity.Відповідають за дії об’єкта чи над об’єктом, зчитують тригери та івенти. Крім цього, можна використовувати для створення графічних ефектів, контролю фізичної поведінки об'єктів.

Transform - кожен об'єкт у сцені має компонент Transform. Він використовується для зберігання та маніпулювання положенням, обертанням та масштабом об’єкта. Кожен компонент Transform може мати батьківський елемент, який дозволяє застосовувати положення, обертання та масштабування ієрархічно.

BlendTree – особливий тип стану в автоматі анімації. BlendTree використовуються для плавного змішування декількох анімацій, включаючи всі їх частини різною мірою.

RPC - метод викликається з клієнта і виконуються на сервері

CMD – викликаються на сервері та виконуються на клієнтах

API - інтерфейс, який визначає взаємодію між безліччю програмних додатків або змішаними апаратно-програмними посередниками

ВСТУП

Інженер програмного забезпечення відповідає за написання гнучкої архітектури, яка дозволяє вносити зміни в проект. Архітектура Unity, що побудована на компонентах (компонент є класом-наслідником від MonoBeheivor) зобов'язана розподілити обов'язки компонентів для їх подальшого налаштування та не створювати залежностей Абстракція-компонент.

Однією з проблем інженера програмного забезпечення є оптимізація запитів до серверу. Оскільки це дозволяє знизити навантаження з сервера та зменшити кількість запитів, що обмежуються, у даному випадку, пропускною здатністю на клієнта - 4000(B/s). Патерн Observer використовується для більшості звернень на сервер, що призводить до зменшення загальної кількості об'єктів серіалізації.

Методи розробки

Unity Animation - повний контроль ваги анімації під час виконання, виклик подій із відтворення анімації, складний автомат ієрархії, кластери графів.

Unity HLAPI - надає доступ до команд, які охоплюють більшість загальних вимог, не турбуючись про деталі реалізації «нижчого рівня». Відповідає за

- Керування мережевим станом гри за допомогою "Network Manager"
- Серіалізування даних
- Здійснення віддалених викликів процедур (RPC) із серверів клієнтам та CMD
- Network Event System

Unity Physics - імітування фізики: прискорення об'єктів, зіткнення, реакція на властивості поверхонь, сила тяжіння та різні інші сили. Можливість кластеризувати фізичні об'єкти

Задачі програмного інженера

- Розробка гнучкої архітектури, яка дозволяє вносити зміни в проект
- Налаштування зв’язку Клієнт-Сервер. Вирішення проблем синхронізації та інтерполювання компонента Transform‘.
- Розробка системи анімацій

Особистий внесок студента:

Розробка архітектури для комп’ютерної гри, що побудована на Unity-components system. Вирішення проблем, пов'язаних з мережами. Створення 2-кластерного графу анімацій

Як результат – цілком готовий проект “CatchUp”

Структура та обсяг роботи

Робота викладена на 68 сторінках друкованого тексту, який складається із вступу, трьох розділів, висновків, списку використаних джерел (23 найменування). Робота містить 12 таблиць, 32 рисунки та 3 додатки, обсягом 12 стор.

РОЗДІЛ 1
АНАЛІЗ ПАРАДИГМ ПРОГРАМУВАННЯ ТА СФЕР ЇХ ЗАСТОСУВАНЬ

1.1 Програмна інженерія на основі компонентів

Програмна інженерія на основі компонентів (CBSE), яку також називають розробкою на основі компонентів (CBD), - це галузь інженерії програмного забезпечення, яка підкреслює розмежування проблем щодо широких функціональних можливостей, доступних у певній програмній системі. Це підхід, що ґрунтується на повторному використанні, до визначення, реалізації та складання незалежних компонентів у системи. Ця практика спрямована на отримання однаково широкого ступеня переваг як у короткостроковій, так і в довгостроковій перспективі.

Індивідуальний компонент програмного забезпечення - це програмний пакет, веб-служба, веб-ресурс або модуль, який інкапсулює набір пов’язаних функцій (або даних).

Що стосується координації, компоненти взаємодіють між собою через інтерфейси. Коли компонент пропонує послуги для решти системи, він приймає наданий інтерфейс, який визначає послуги, які можуть використовувати інші компоненти, і спосіб їх використання. Клієнту не потрібно знати про внутрішню роботу компонента (реалізації), щоб скористатися ним. Цей принцип призводить до того, що компоненти є інкапсульованими.

Для доступу до компонента або спільного використання в контексті виконання або мережевих посилань, використовуються серіалізація.

Багаторазове використання є важливою характеристикою високоякісного програмного компонента.

Компоненти можуть створювати або споживати події та можуть використовуватися для архітектур, керованих подіями (EDA).

1.2 Архітектура, керована подіями

Архітектура, керована подіями (EDA) - це парадигма архітектури програмного забезпечення, що сприяє виробленню, виявленню, споживанню та реагуванню на події.

Цей архітектурний шаблон застосовується у ПЗ, в тому числі, і для оптимізації кількості звернень до серверу наступним чином:

```
Class BombGiver: NetworkBehaviour
{
  [SyncVar (hook = nameof(OnActiveSet))]
  public bool isActive;
    public void OnActiveSet(bool a)
    {
        bomb.SetActive(a);
        CmdResetTimer();
    }
}
```

Клас BombGiver відповідає за передачу ролі гравця. Булева змінна isActive синхронізується з сервером. При її зміні, сервер одночасно повідомить усіх клієнтів про подію. Також, до змінної прив’язаний метод OnActiveSet, що виконається при зміні значення isActive.

Побудова систем навколо керованої подіями архітектури спрощує горизонтальну масштабованість в розподілених обчислювальних моделях і робить їх більш стійкими до відмов. Це пов’язано з тим, що стан програми можна скопіювати через кілька паралельних знімків для високої доступності. Нові події можуть бути ініційовані в будь-якому місці, але, що ще важливіше, поширюватимуться по мережі сховищ даних, оновлюючи їх по мірі надходження.

1.3 Принципи компонентно-орієнтованої архітектури

1. Низьке зчеплення

зчеплення - це міра того, наскільки міцно один елемент пов'язаний, знає чи покладається на інші елементи. Низький зв'язок - це оцінна модель, яка диктує, як розподілити відповідальність за наступні переваги:

- менша залежність між класами,
- зміни в одному класі, що має менший вплив на інші класи,
- вищий потенціал повторного використання.

2. Висока згуртованість

Висока згуртованість - це оцінна модель, яка намагається зберегти об’єкти належним чином сфокусованими, керованими та зрозумілими. Висока згуртованість, використовується для підтримки низького зчеплення. Висока згуртованість означає, що обов'язки даного елемента тісно пов'язані та високо зосереджені. Розбиття програм на класи та підсистеми є прикладом діяльності, що підвищує цілісні властивості системи.

3. Залежності компонента від абстракції

Принцип залежностей компонента від абстракції диктує наступні правила:

A. Модулі високого рівня не повинні залежати від модулів низького рівня. І те, і інше повинно залежати від абстракцій (наприклад, інтерфейсів).

B. Абстракції не повинні залежати від компонентів. Компоненти (конкретні реалізації) повинні залежати від абстракцій.

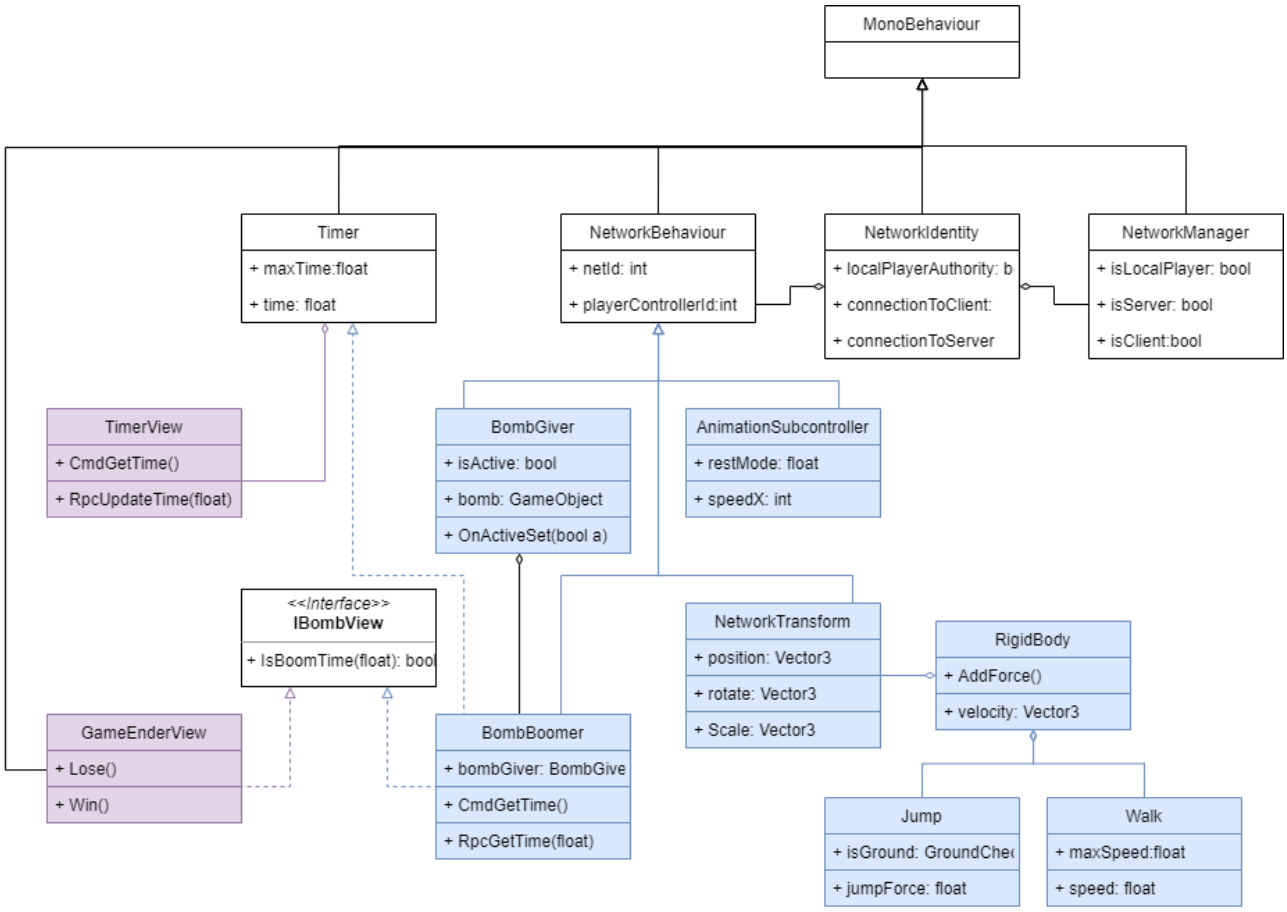
A. Яскравим прикладом цього може слугувати інтерфейс IBombView. IBombView реалізують GameEnderView та BombBoomer. Цей інтерфейс абстрагує поняття GUI від конкретного компонента BombBoomer, що надає можливість створювати інший вид GUI не змінюючи компонент BombBoomer.

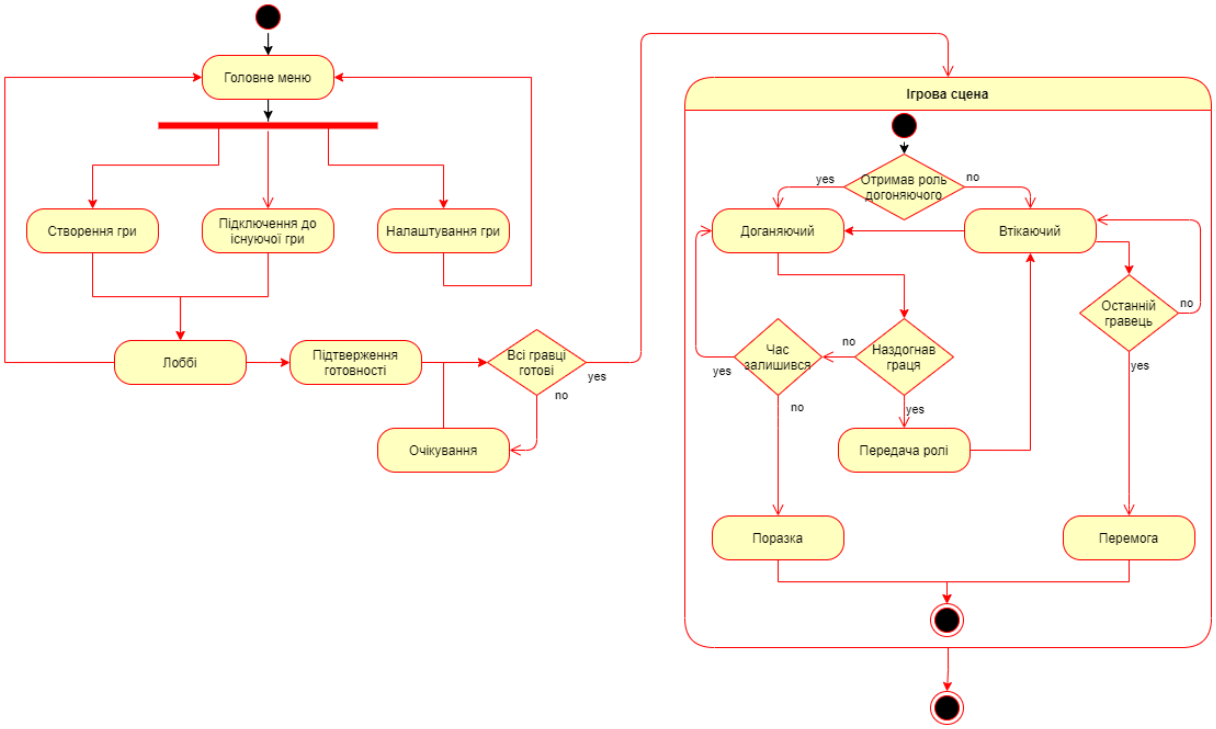
B. Залежності абстракції від компонента не повинно бути в програмному забезпеченні. Компоненти – виконавчі класи, що наслідуються від класу MonoBeheivor. Це окремий модуль, що повинен мати низький показник зчеплення. Прикладом виконання цього принципу може слугувати агрегація абстракції Timer. TimerView та BombBoomer агрегують Timer та залежать від нього. Timer, хоч і є наслідником MonoBeheivor – не є конкретною деталлю.

1.4 Висновки до розділу

Проект виконаний з дотриманням всіх вимог програмної інженерії на основі компонентів. З використанням патерну Observer.

Проект відповідає стандарту бінарного інтерфейсу “Component Object Model” (COM)




Головне меню
Створення гри
Підключення до існуючої гри
Налаштування гри
Лоббі
Підтвердження готовності
Всі гравці готові
yes
no
Очікування
Ігрова сцена
Отримав роль догоняючого
yes
no
Доганяючий
Втікаючий
Останній гравець
no
yes
Час залишився
no
yes
Наздогнав граця
no
yes
Передача ролі
Поразка
Перемога

РОЗДІЛ 2
РЕАЛІЗАЦІЯ ЗВ’ЯЗКУ КЛІЄНТ-СЕРВЕР ПРИ РОЗРОБЦІ КОМП’ЮТЕРНОЇ ГРИ

2.1 High Level API

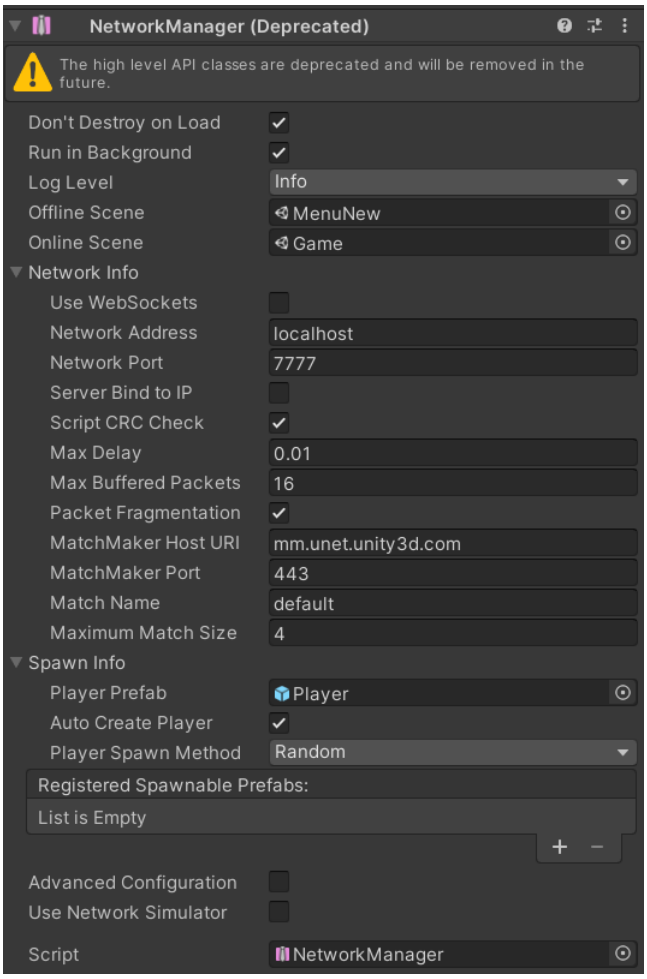
Unity’s multiplayer High Level API (HLAPI) - це система для створення багатокористувацьких можливостей для ігор Unity. Він побудований на нижньому рівні та обробляє багато загальних завдань, необхідних для багатокористувацьких ігор. HLAPI - дозволяє одному з учасників бути одночасно і клієнтом, і сервером, тому виділений сервер не потрібен.

2.2 NetworkManager

NetworkManager - це основний компонент управління багатокористувацькою грою.

Для NetworkManager редактор дозволяє налаштувати та контролювати багато речей, пов’язаних з мережею.

Network Manager реалізований повністю за допомогою API високого рівня (HLAPI), тому все, що він робить, також доступне за допомогою сценаріїв. Компонент Network Manager об'єднує багато корисних функцій в одне місце.



2.3 Управління ігровим станом

Мережева багатокористувацька гра може працювати в трьох режимах - як клієнт, як виділений сервер або як “Хост”, який одночасно є і клієнтом, і сервером.

Для написання кастомного меню також був використаний NetworkManager а саме такі його методи як:

NetworkManager.StartClient() – для створення обьекту, що представляє клієнта.

NetworkManager.StartServer() – для запуску нового серверу.

NetworkManager.StartHost() - для запуску "хоста" - сервер і клієнт в одному об’єкті.

Для цих методів використовуються поля networkAddress та networkPort з NetworkManager, як адресу для підключення та відкриття з’єднання.

Клієнт, повернутий із StartHost (), є спеціальним "локальним" клієнтом, який здійснює зв'язок із процесорним сервером за допомогою черги повідомлень замість реальної мережі. Але майже у всіх інших випадках до нього можна ставитись як до звичайного клієнта.

2.4 Налаштування мережевої адреси та порту в компоненті NetworkManager

У якому б режимі гра не починалася (клієнт, сервер чи хост), використовуються властивості networkAddress та networkPort. У клієнтському режимі гра намагається підключитися до вказаної адреси та порту. У режимі сервера або хосту гра прослуховує вхідні з'єднання на вказаний порт.

Під час розробки гри корисно вказати фіксовану адресу та налаштування порту в цих властивостях для Matchmaker, що буде використана, щоб знайти матчі в Інтернеті для підключення

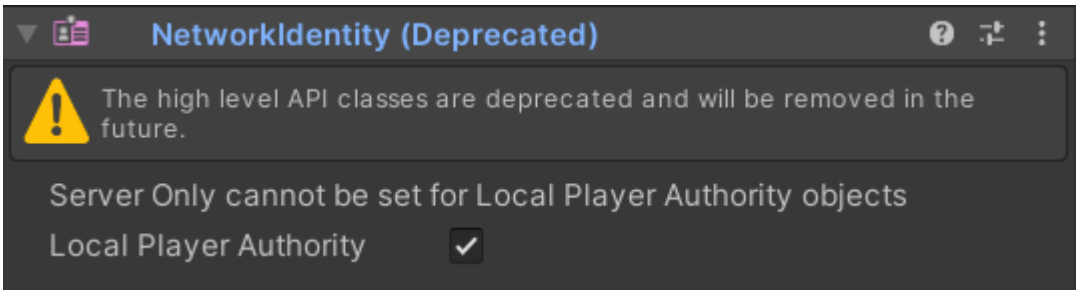
2.5 Управління Spawn

У проекті був використаний NetworkManager для управління Spawn.

.

У даному проекті гравець представлений у вигляди Prefab, тому ми маємо змогу передавити його у якості значення для змінної. Використовуючи поле Player Prefab гравець GameObject автоматично створюється з цього Prefab для кожного користувача в грі. Це стосується локального програвача на розміщеному сервері та віддалених програвачів на віддалених клієнтах.

До префабу обов’язково повинен бути прєднаним компонент Network Identity.



- Server Only - гарант того, що Unity створює GameObject лише на сервері, а не на клієнтах.
- Local Player Authority - надання клієнту, який володіє GameObject, повноважний мережевий контроль цього GameObject.

2.6 Використання віртуальних методів NetworkManager

У класі NetworkManager є віртуальні функції, що використовуються для налаштування мережі. Вони дають змогу, створивши власний похідний клас, який успадковує від NetworkManager, реалізувати ці функції:

- OnClientEnterLobby() - викликається на всіх об'єктах гравця при вході до серверу.
- OnClientExitLobby - викликається на всіх об'єктах гравця під час виходу з серверу.
- OnClientReady - викликається на клієнтах, коли LobbyPlayer перемикається між станами готовим чи не готовим почати гру.
- та інші…

Для реалізації ігрового лоббі був створенний клас CustomNetworkLobbyPlayer та CustomNetworkLobbyManager. Для створення події на підключення був перезаписаний віртуальний метод з класу CustomNetworkLobbyPlayer(наслідується від NetworkLobbyPlayer).

```
public override void OnClientEnterLobby()
{
    base.OnClientEnterLobby();

    Debug.Log("Client is Enter to Lobby");


    var slots = FindObjectOfType<CustomNetworkLobbyManager>().lobbySlots;
    var playerList = FindObjectOfType<LobbyController>();
    playerList.ShowLobbyPlayer(slots);
}
```

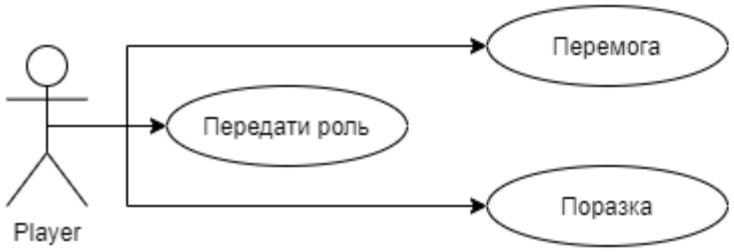
2.7 Збереження інформації про гравця до ігрового стану

Для збереження інформації про гравця до ігрового стану, тобто в стані Меню, було прийняте рішення представляти гравця об’єктом NetworkLobbyPlayer. NetworkLobbyPlayer зберігає такі дані як

- Ім’я гравця
- Готовність розпочати гру

Оскільки однією з задач була розробка гнучкої архітектури, то був створений клас, що наслідується від NetworkLobbyPlayer. У разі потреби, клас можна модифікувати/наслідувати не зачіпляючи інших компонентів програми.

У ігровому стані програми, об’єкт замінюється на Player, що відповідає за поведінку ігрового персонажа.



2.8 Висновки до розділу

У ході реалізації зв’язку клієнт-сервер було проаналізовано існуючі реалізації зв’язку клієнт-сервер. Було вирішено проблеми реалізації зв’язку клієнт-сервер пов’язані з відключенням гравця, через нестабільне інтернет підключення

Було створено власні версії класів NetworkLobbyPlayer NetworkLobbyManager, що дозволило зберігати вузькоспеціалізовану інформацію та дозволило створити власну обгортку на ігрове лобі.

РОЗДІЛ 3
КЛАСТЕРИЗАЦІЯ ГРАФУ АНІМАЦІЙ

Анімаційний компонент використовується для відтворення анімації.

Анімаційні кліпи призначаються компоненту анімації та контролюють відтворення зі свого сценарію. Система анімації в Unity базується на

- змішуванні анімацій
- додаванні анімацій
- кластеризації

AnimationState використовується для зміни рівня анімації, зміни швидкості відтворення та для безпосереднього контролю над змішуванням

3.1 Анімаційна система

Система анімації забезпечує:

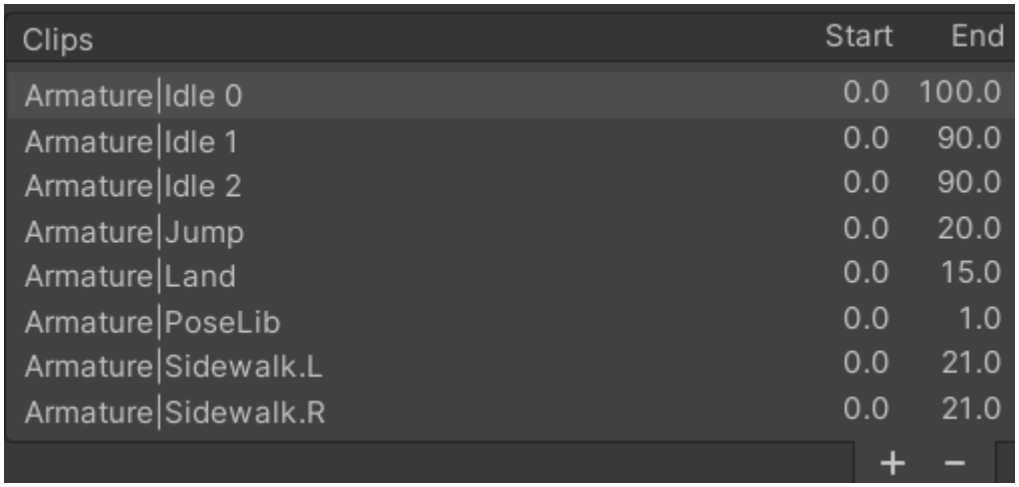
- Простий робочий процес та налаштування анімації для всіх елементів включаючи об’єкти, символи та властивості.
- Підтримка імпортованих анімаційних кліпів та анімації
- Спрощений робочий процес для вирівнювання анімаційних кліпів.
- Попередній перегляд анімаційних кліпів, переходів та взаємодії між ними.
- Управління складною взаємодією між анімацією за допомогою інструменту візуального програмування.
- Анімація різних частин тіла з різною логікою.
- Особливості кластеризації та маскування

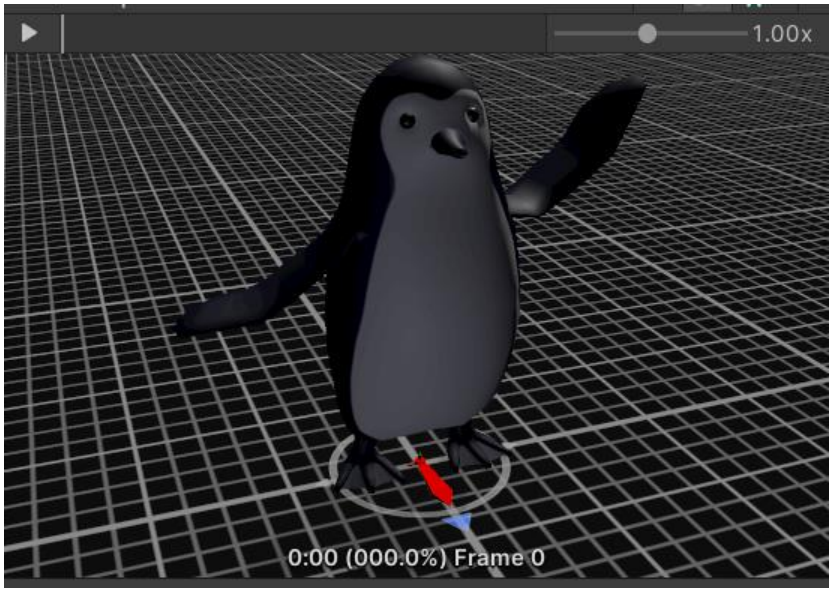
3.2 Робочий процес анімації

Система анімації Unity базується на концепції анімаційних кліпів, які містять інформацію про те, як певні об’єкти повинні змінювати своє положення, обертання чи інші властивості з часом. Кожен кліп можна сприймати як єдиний лінійний запис.

При створенні анімації для персонажа CatchUp використовувався Blender. У даному редакторі були створені наступні анімації:

- Idle0 – Анімація “Заглушка”
- Idle1 – перша анімація відпочинку
- Idle2 – друга анімація відпочинку
- Jump – анімація стрибку
- Land – анімація приземлення
- PoseLib - анімація позування
- SideWalk.L – анімація бокової ходьби(ліво)
- SideWalk.R - анімація бокової ходьби(право)





3.3 Animator Controller

Анімаційні кліпи організовуються у структуровану схожу на блок-схему систему, яка називається Animator Controller. Він відстежує який кліп повинен відтворюватися в даний момент, і коли анімація повинна змінюватися або поєднуватися.

Animator Controller використовує набір параметрів для переходу між станами. Наявні параметри:

- SpeedX(float) – швидкість персонажа по осі X
- SpeedY(float) - швидкість персонажа по осі Y
- RestMode(float) – вибір анімації відпочинку (керується Unity.Random)
- IsMoving(bool) – якщо персонаж рухається: true
- IsFlying(bool) - якщо персонаж в повітрі: true
- IsJumping(Trigger) - якщо гравець натиснув “стрибок”: true

Керує цим процесом клас AnimationSubcontroller, розбір одного з методів:

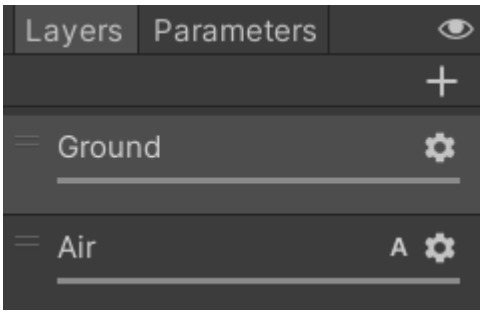
```
    private void Walking()
    {
        if (gc.onGround)
        {
            animator.SetFloat(speedX, Input.GetAxis("Horizontal"));
            animator.SetFloat(speedY, Input.GetAxis("Vertical"));
        }
        else
        {
            animator.SetFloat(speedX, 0);
            animator.SetFloat(speedY, 0);
        }
    }
```

Метод Walking() зчитує дані з приладу вводу даних (можливість обробляти дані з будь-яких приладів вводу даних) та встановлює значення параметрів SpeedX, SpeedY. Якщо ж персонаж знаходиться не на землі – включає анімацію ходьби персонажа, інтерполюючи її з анімацією падіння та стрибку.

3.4 Кластери анімацій

При розробці даного проекту виникла проблема “накладання” анімацій. Було прийнято рішення кластеризувати окремі частини графів. У ході кластеризації були виділені наступні кластери:

- Ground(відповідає за анімацію персонажа на землі)
- Air(відповідає за анімацію персонажа у повітрі)



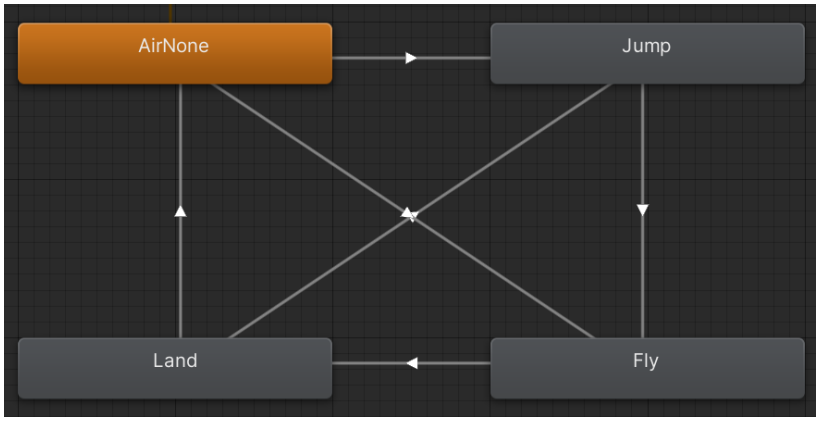
Кластер Air є Аддитивним, на відміну від Ground(Override). Таким чином вдалось запобігти “накладання” анімацій.

3.5 Граф анімацій

Перехід станів на графі виконується за допомогою параметрів. На кластері Air наявні наступні стани анімацій:

- AirNone – виконується у якості “заглушки”, коли кластер не є активним
- Jump – виконується при натисканні гравцем функціональної клавіши “стрибок”
- Fly – виконується у випадку, коли персонаж знаходиться в повітрі
- Land – виконується у випадку, коли відбувається перехід від стану Fly до AirNone

Оскільки граф орієнтований, то переходу станів відбуваються у заданій послідовності. Перехід зі стану Jump до стану Fly є можливим, але не у зворотному напрямку, оскільки функція Jump не повинна бути доступна не на землі.

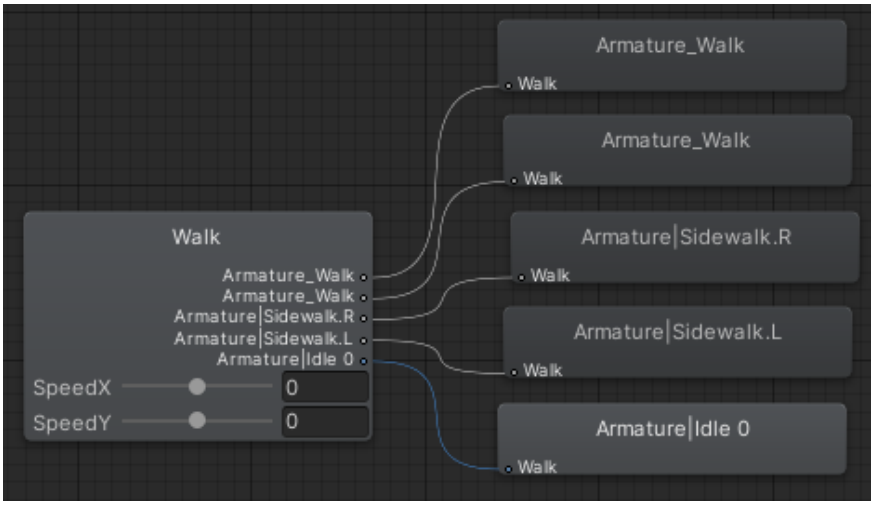


3.6 Blend tree. Інтерполяція анімацій

Blend tree використовуються для плавного змішування декількох анімацій, включаючи всі їх частини різною мірою. Сума, яку кожен з рухів вносить у кінцевий ефект, контролюється за допомогою параметра змішування, який є лише одним із числових параметрів анімації, що використовуються для зв'язку між сценаріями та AnimatorController.

Даний Blend tree використовує наступні параметри:

- SpeedX
- SpeedY



При їх зміні виконується інтерполювання різних анімацій. Таким чином можна досягти плавного переходу між початком ходьби та відпочинком, або одночасного програвання анімації ходьби вперед та вбік.

3.7 Висновок до розділу

У ході проведення роботи з анімацією було проаналізовано існуючі методи програвання анцісацій. Було вирішено проблеми програвання анцімацій пов’язані з:

- Співвідношенням швидкості анімаціїї до швидкості персонажа
- “накладанням” анімацій
- Синхронізації фізичного відштовхування з анімацією відштовхування
- Залежністю анімації від компонента Walk

Було створено двокластерний граф анцімацій. BlendTree, що керується за допомогою двох параметрів

РОЗДІЛ 4
УПРАВЛІННЯ ПЕРСОНАЖЕМ

4.1 способи переміщення тіл у Unity

Unity передбачає наступні типи переміщень:

1. Зміна компонента Transform

   Кожен GameObject, що є на сцені проекту має компонент Transform. компонент Transform відповідає за збереження інформації про місце положення обьякта. Має наступні моля:

   a. Position: Vector3
   b. Rotation: Vector3
   c. Scale: Vector3

   За допомогою трьох полів можна описати точне положення компонента на сцені.
2. Character controller – компонент, що відповідає за переміщення персонажа. Має наступні поля для редагування:
   a. Обмеження нахилу

      Обмежує колайдер лише тими схилами, які є менш крутими (у градусах), ніж зазначене значення.
   b. Зсув кроку

      Персонаж активізуватиме сходи тільки в тому випадку, якщо він буде ближче до землі, ніж зазначене значення. Це не повинно бути більше висоти Character controller, або це призведе до помилки.
   c. Ширина обкладинки

      Два колайдери можуть проникати один в одного так глибоко, як їх ширина шкіри. Більша ширина шкіри зменшує коливання. Низька ширина шкіри може призвести до застрягання персонажа. Хорошим параметром є значення 10% радіуса.
   d. Мінімальна відстань переміщення

      Якщо Character controller спробує перейти нижче вказаного значення, він взагалі не буде рухатися. Це можна використовувати для зменшення коливання. У більшості випадків це значення слід залишати на 0.
   e. Центр

Це компенсує капсульний колайдер у світовому просторі і не вплине на те, як обертається Character controller.

f. Радіус

Довжина радіуса капсульного колайдера. По суті, це ширина колайдера.

g. Зріст

Зміна цього масштабу призведе до масштабування колайдера вздовж осі Y як в позитивному, так і в негативному напрямках.

3. RigidBody

RigidBody дозволяє об'єктам взаємодіяти з фізикою. Для реалістичного руху твердих тіл на останніх впливає сила обертання та інші сили. Будь-який ігровий об'єкт повинен містити тверде тіло, яке підлягає гравітації, діяти відповідно до сил, призначених сценарієм, або взаємодіяти з іншими об'єктами через фізичний двигун NVIDIA PhysX.

4.2 Проблеми переміщення

1. Зміна компонента Transform або CharacterController

За допомогою цих компонентів можна досягті чіткого переміщення обьекта по координатній системі. Реалізація стрибку потребує програмування більш низького рівня.

```
    private float CalculateJumpSpeed(float jumpHeight, float gravity)
    {
        return Mathf.Sqrt(2 * jumpHeight * gravity);
    }
```

Але цей спосіб не передбачає реакції з характеристиками поверхні, тому було прийнято рішення відмовитись від управління за допомогою зміни компонента Transform або CharacterController.

2. RigidBody

Управління здійснюється за допомогою метода AddForce() що надає “силу” до об’єкта, у фізичному її розумінні. Оскільки у такому переміщенні важлива кількість ітерацій у секунду – виконується метод у тілі метода FixedUpdate(). Так можна уникнути зміни швидкості через різницю у кількості кадрів в секунду.

Так як AddForce() не прибирає сили від середовища, наприклад сили тяжіння або реакції опори, то він є гарним рішенням для управління, що поєднане з впливом середовища.

4.3 Проблеми прискорення

Хоча у проекті й викликається AddForce() тільки у тілі метода FixedUpdate(), але проблема прискорення на цьому не вирішена. Оскільки для початку руху потрібно більше сили ніж для його підтримки – настає питання: “Як урівняти сили для миттєвого руху та запобігти прискоренню на стадії його підтримки?”.

Приклад коду, що відповідає за переміщення персонажа:

```
Vector3 b = new Vector3(Input.GetAxis("Horizontal"), 0,
Input.GetAxis("Vertical"));
if (b.magnitude > 1)
    b = b.normalized;

if (Input.GetAxis("Vertical") < 0)
    b.x /= 2;
b = transform.TransformDirection(b) * speed;

var velocityChange = b - rb.velocity;
velocityChange.y /= 2;

rb.AddForce(b * speed, ForceMode.VelocityChange);
```

```
Vector3 b = new Vector3(Input.GetAxis("Horizontal"), 0,
Input.GetAxis("Vertical"));
```

Отримання операцій переміщення абстраговане від пристрою вводу. Це дозволяє використовувати різні види вводу не змінюючи компоненти, що відповідають за переміщення персонажа.

```
if (b.magnitude > 1)
    b = b.normalized;
```

У випадку, якщо довжина вектора переміщення більша за одиницю, то вектор потрібно нормалізувати, тобто зробити його довжину рівної одиниці. Так вдалось запобігти зростанню швидкості, що виникає у разі ходьби по діагоналі(1,0,1).

```
b = transform.TransformDirection(b) * speed;
var velocityChange = b - rb.velocity;
```

Повертаючись до питання “Як урівняти сили для миттєвого руху та запобігти прискоренню на стадії його підтримки?”.

Було прийнято рішення брати поточну швидкість об'єкту та змінювати її в залежності від поставленої “Максимальної швидкості”. Так можна уявити швидкість у якості одномірної величини. Якщо ж поточна швидкість більша за максимальну дозволену, то потрібно провести операцію віднімання поточної швидкості від максимальної швидкості.

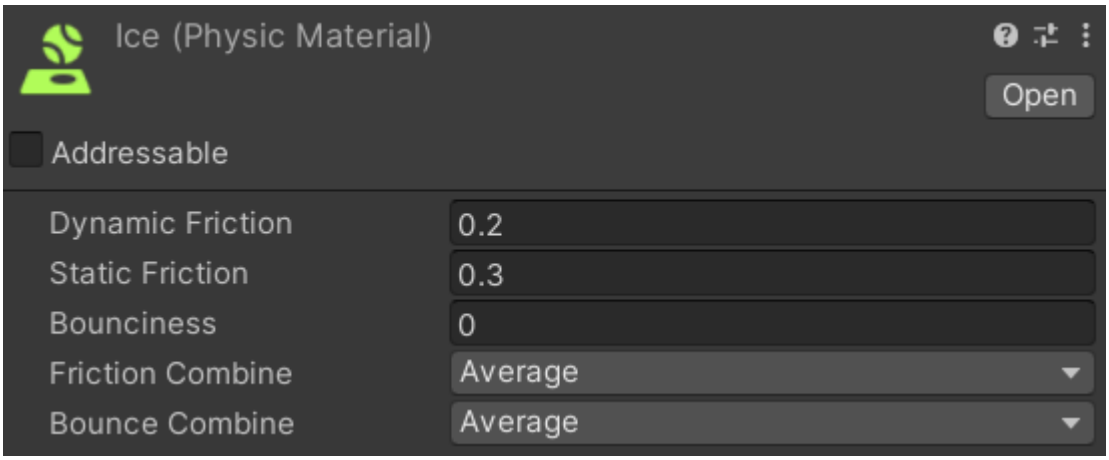
```
        rb.AddForce(b * speed, ForceMode.VelocityChange);
```

4.4 Реакція поверхні

Фізика матеріал використовується для налаштування впливу тертя і відмов об'єктів при зіткненні.

Тертя - це значення, яке відповідає за запобігання потирання поверхонь один проти одного. Тертя може бути двох видів, динамічним і статичним. Статичне тертя використовується, коли об'єкт нерухомий. Це не дозволить такому об'єкту рухатися без впливу зовнішніх сил.

Оскільки управління у грі CatchUp не є звичайним, а саме базується на різких стрімких рухах з великим періодом очікування, то важливу роль відіграє імітування льоду, як основний тип поверхонь.



Так за допомогою Physic Material вдалось досягти доволі близького до справжнього льоду імітації. У стани спокою персонаж не буде рухатись, оскільки на нього діє статичне тертя, що дорівнює коефіцієнту 0.3. Завдяки цьому була уникнута така проблема як постійне змінення компонента Transform, що тягне за собою потребу синхронізації з клієнтами та сервером, загружаючи інтернет трафік і розрахункові потужності серверу.

У стані переміщення на персонажа починає діяти динамічне тертя, сповільнюючи персонажа кожну ітерацію фізичної обробки сцени. Завдяки

цьому досягається плавний перехід між переміщенням та зупинкою персонажа.

Модель тертя, що використовується двигуном Nvidia PhysX, налаштована на продуктивність і стабільність моделювання, і не обов'язково представляє близьке наближення фізики реального світу. Зокрема, контактні поверхні, які більші за одну точку будуть обчислюватися як такі, що мають дві контактні точки, і маючи сили тертя вдвічі більші, ніж у фізиці реального світу.

4.5 Event для зв’язку з анімаціями

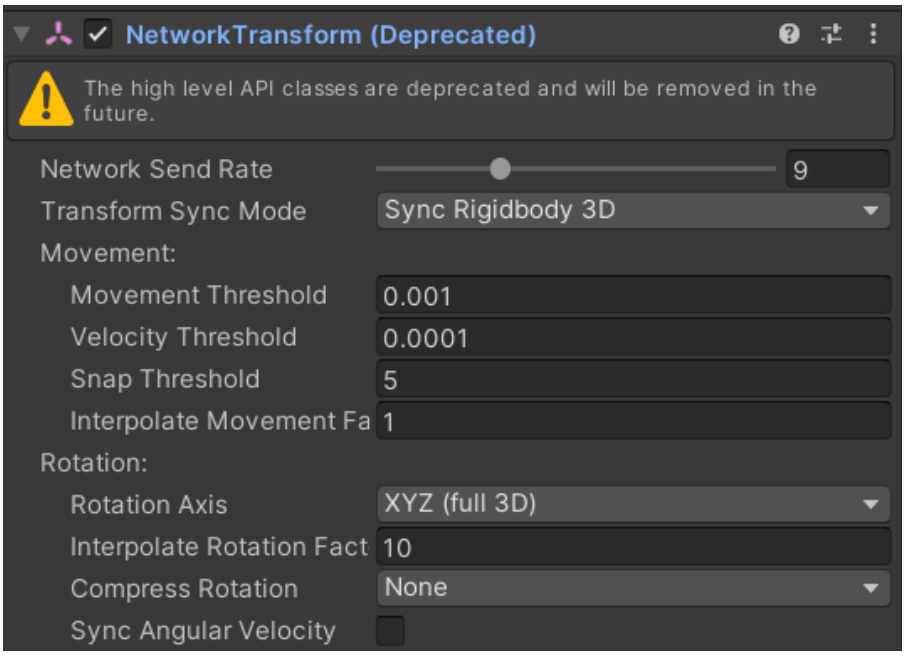
Анімація переміщення повинна не тільки брати поточну швидкість по осям координат X та Y, а я відповідати “ривкам” що імітують відштовхування від поверхні. За кожен ривок відповідає компонент Walk, саме він визначає проміжок часу між ривками, а отже і він повинен відповідати за виклик методів. Для запобігання порушення принципів зв’язності – Walk реалізує інтерфейс IAnimator. Так вдалось абстрагувати анімації, що призвело до збільшення гнучкості архітектури проекту. Таким чином можна змінювати анімації(контролер яких, реалізує інтерфейс IAnimator) незважаючи на компонент Walk.

4.6 Синхронізація переміщення

Network Transform синхронізує рух і обертання GameObject по всій мережі.

Цей компонент враховує повноваження, тому локальний програвач GameObjects (які мають місцеві повноваження) синхронізують свою позицію з клієнта на сервер, а потім з іншими клієнтами. Інші GameObjects (із повноваженнями сервера) синхронізують свою позицію з сервера на клієнтів.

GameObject із компонентом Network Transform також повинен мати компонент Network Identity. Коли ви створюєте компонент Network Transform на GameObject, Unity також створює компонент Network Identity на цьому GameObject, якщо його ще немає.



Оскільки на етапі вибору методу переміщення персонажа був вибраний метод Rigidbody, то й синхронізація з сервером буде проводитись по Rigidbody. Для економії трафіку був збільшений мінімальний поріг для проведення синхронізації, оскільки гра побудована на різких поштовхах де не є важливим змінення положення на малих масштабах. Також була збільшена швидкість інтерполяції повороту через високу динамічність CatchUp.

Серіалізація компонента Rigidbody виконується за наступними параметрами:

4.7 Висновки до розділу

У ході проведення роботи з управління персонажем було проаналізовано існуючі методи управління та виявлено найбільш підходящий до даного типу управління. Було вирішено проблеми переміщення пов’язані з:

- Реакцією на навколишнє середовище
- Зміною швидкості, в залежності від кількості кадрів в секунду
- Прискоренням персонажа, через неявну систему реакції поверхні з фізичною моделлю персонажа.
- Синхронізації фізичного відштовхування з анімацією відштовхування
- Залежністю анімації від компонента Walk
- Сповільненим поворотом моделі персонажа іншого гравця

ВИСНОВОК

Був створений проект “CatchUp” за мотивами міжнародного чемпіонату “World Chase Tag”. На даний момент проект є цілком готовий до виходу на ринок.

Проект виконаний з дотриманням всіх вимог програмної інженерії на основі компонентів. З використанням патерну Observer.

Проект відповідає стандарту бінарного інтерфейсу “Component Object Model” (COM)

Було вирішено проблеми реалізації зв’язку клієнт-сервер пов’язані з відключенням гравця, через нестабільне інтернет підключення. Створено власні версії класів NetworkLobbyPlayer NetworkLobbyManager, що дозволило зберігати вузькоспеціалізовану інформацію та дозволило створити власну обгортку на ігрове лоббі.

У ході проведення роботи з анімацією було проаналізовано існуючі методи програвання анцісацій. Було вирішено проблеми програвання анцімацій пов’язані з:

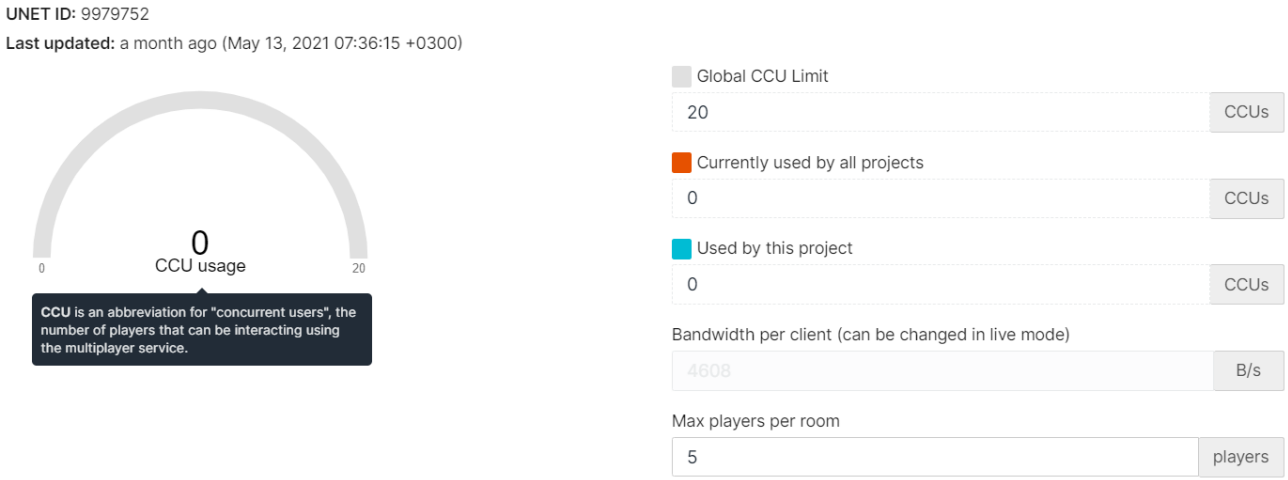
- Співвідношенням швидкості анімаціїї до швидкості персонажа
- “накладанням” анімацій
- Синхронізації фізичного відштовхування з анімацією відштовхування
- Залежністю анімації від компонента Walk

Було створено двокластерний граф анцімацій. BlendTree, що керується за допомогою двох параметрів.

У ході проведення роботи з управління персонажем було проаналізовано існуючі методи управління та виявлено найбільш підходящий до даного типу управління. Було вирішено проблеми переміщення пов’язані з:

- Реакцією на навколишнє середовище
- Зміною швидкості, в залежності від кількості кадрів в секунду
- Прискоренням персонажа, через неявну систему реакції поверхні з фізичною моделлю персонажа.
- Сповільненим поворотом моделі персонажа іншого гравця

Вирішені проблеми пов’язані з передачею повідомлень на сервер та їх оптимізацією. Завдяки цьому “CatchUp” не потребує великих обчислювальних потужностей на стороні сервера та пропускної здатності на клієнта (B/s).



СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Brad J. Cox, Andrew J. Novobilski (1991). Object-Oriented Programming: An Evolutionary Approach. 2nd ed. Addison-Wesley, Reading ISBN 0-201-54834-8
2. Bertrand Meyer (1997). Object-Oriented Software Construction. 2nd ed. Prentice Hall.
3. Pecinovsky, Rudolf (2013). OOP – Learn Object Oriented Thinking & Programming. Bruckner Publishing. ISBN 978-80-904661-8-0.
4. Muhammad Umair (2018-02-26). "SOLID, GRASP, and Other Basic Principles of Object-Oriented Design"
5. Martin, Robert (2018). Clean Architecture: A Craftsman's Guide to Software Structure and Design. p. 58. ISBN 9780134494166.
6. Unity Multiplayer and Networking [Електронний ресурс] - Режим доступу до ресурсу:Unity - Manual: Multiplayer and Networking (unity3d.com)
7. Unity User Manual 2021.2 Alpha [Електронний ресурс] - Режим доступу до ресурсу: https://docs.unity3d.com/2021.2/Documentation/Manual/UnityManual.html

ДОДАТКИ

Додаток А

```
public class AnimationSubcontroller : NetworkBehaviour
{
    private int isFlying = Animator.StringToHash("IsFlying");
    private int isMoving = Animator.StringToHash("IsMoving");
    private int isJumping = Animator.StringToHash("IsJumping");

    private int idle1 = Animator.StringToHash("idle1");
    private int idle2 = Animator.StringToHash("idle2");

    private int speedX = Animator.StringToHash("SpeedX");
    private int speedY = Animator.StringToHash("SpeedY");


    private float restMode = Animator.StringToHash("RestMode");

    private PlayerController player;
    private Animator animator;
    private NetworkAnimator netAnimator;

    private Random rn;

    private void Awake()
    {
        netAnimator = GetComponent<NetworkAnimator>();
        netAnimator.SetParameterAutoSend(0, true);
        netAnimator.SetParameterAutoSend(1, true);
        netAnimator.SetParameterAutoSend(2, true);
        netAnimator.SetParameterAutoSend(3, true);
        netAnimator.SetParameterAutoSend(4, true);
        netAnimator.SetParameterAutoSend(5, true);



        animator = GetComponent<NetworkAnimator>().animator;
        rn = new Random();

        player = GetComponent<PlayerController>();
    }
    private void Update()
    {
        if (isLocalPlayer)
        {
            Resting();


            Walking();


            Jumping();
            Flying();
        }
    }
    private void Resting()
    {
```

```
        var stateInfo = animator.GetCurrentAnimatorStateInfo(0);
        if (stateInfo.IsName("Idle0"))
        {
            switch (rn.Next(2))
            {
                case 0:
                    animator.SetTrigger(idle1);
                    animator.ResetTrigger(idle2);
                    break;

                case 1:
                    animator.SetTrigger(idle2);
                    animator.ResetTrigger(idle1);
                    break;
            }
        }
    }
    private void Walking()
    {
        if (player.isGrounded)
        {
            animator.SetFloat(speedX, Input.GetAxis("Horizontal"));
            animator.SetFloat(speedY, Input.GetAxis("Vertical"));
        }
        else
        {
            animator.SetFloat(speedX, 0);
            animator.SetFloat(speedY, 0);
        }

    }

    private void Jumping()
    {
        if (Input.GetButtonDown("Jump") && player.isGrounded)
        {
            netAnimator.SetTrigger(isJumping);
            animator.SetTrigger(isJumping);
        }
        else
            animator.ResetTrigger(isJumping);
    }
    private void Flying() => animator.SetBool(isFlying, !player.isGrounded);
}
```

Додаток Б

```
public class PlayerController : NetworkBehaviour
{
    [SerializeField] private float speed = 5.0f;
    [SerializeField] private float speedMax = 10.0f;
    [SerializeField] private float jumpForce = 200.0f;
    [SerializeField] private LayerMask layerMask;
    [SerializeField] private Transform target;

    [SerializeField] private float waitTime = 0.5f;

    private bool isMoveTime = true;
```

```
    private Rigidbody rb;
    public bool isGrounded;
    private bool isJumpKeyDown;

    private void Awake()
    {
        Cursor.lockState = CursorLockMode.Locked;
        rb = gameObject.GetComponent<Rigidbody>();
    }

    private void Start()
    {
        if (isLocalPlayer)
        {
            target.gameObject.SetActive(true);
        }
    }

    private void Update()
    {
        if (!isLocalPlayer) return;

        MouseControll();

        if (Input.GetButtonDown("Jump"))
            isJumpKeyDown = true;
    }

    private void FixedUpdate()
    {
        if (!isLocalPlayer) return;

        isGrounded = Physics.Raycast(transform.position,
transform.TransformDirection(Vector3.down), 0.1f, layerMask);
        if (isGrounded)
            Move();

        Jump();
    }
    private void Move()
    {

        if (rb.velocity.magnitude >= speedMax)
            return;

        Vector3 direction = new Vector3(Input.GetAxis("Horizontal"), 0,
Input.GetAxis("Vertical")).normalized;
        Debug.Log(direction);

        if (direction.magnitude == 1 && isMoveTime)
        {
            rb.AddRelativeForce(direction * speed, ForceMode.Impulse);
            isMoveTime = false;

            StartCoroutine(WaitMoveTime());
        }
```

```
        #region
        /*
Vector3 b = new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
if (b.magnitude > 1)
    b = b.normalized;

if(Input.GetAxis("Vertical") < 0)
    b.x /=2;
b = transform.TransformDirection(b) * speed;

var velocityChange = b - rb.velocity;
velocityChange.y /= 2;

rb.AddForce(b*speed, ForceMode.VelocityChange);
*/

        /* Move Velocity
        b = transform.TransformDirection(b) * speed;
        rb.velocity = b + Vector3.up * rb.velocity.y;
        */
        #endregion
    }
    private IEnumerator WaitMoveTime()
    {
        yield return new WaitForSeconds(waitTime);
        isMoveTime = true;
        yield return null;
    }

    private void Jump()
    {
        if (isJumpKeyDown)
        {
            if (isGrounded)
            {
                rb.AddForce(jumpForce * Vector3.up);
            }
            isJumpKeyDown = false;
        }
    }


    private void MouseControll()
    {
        float mouseX = Input.GetAxis("Mouse X");
        float mouseY = Input.GetAxis("Mouse Y");

        const int max = 80;
        const int min = -40;
        var angle = target.rotation.eulerAngles.x;

        if (angle > 180)
            angle -= 360;

        var tooLow = angle < min && mouseY < 0;
        var tooHigh = angle > max && mouseY > 0;
        var normal = angle >= min && angle <= max;
```

```
        transform.Rotate(Vector3.up * mouseX);
        if (tooLow || tooHigh || normal)
            target.Rotate(-Vector3.right * mouseY);
    }

}
```

Додаток В

```
public class BombCreator : MonoBehaviour
{
    void Update()
    {

        var bomb = FindObjectOfType<BombGiver>();
        if (bomb != null)
        {
            var allBombs = FindObjectsOfType<BombGiver>();
            if(allBombs.All(x => !x.isActive))
                bomb.isActive = true;

            Destroy(this);
        }

    }
}
```

Додаток Г

```
public class BackController : MonoBehaviour
{
    private PageHistory pageHistory;
    private void Awake()
    {
        pageHistory = FindObjectOfType<PageHistory>();
        pageHistory.OnHistoryChanged += Display;
    }

    private void Start() => Display();
    public void Back()
    {
        GetFirstChild(pageHistory.GoBack())
            .SetActive(false);

        GetFirstChild(pageHistory.Get())
            .SetActive(true);
    }

    private void Display()
    {
        if (pageHistory.Count() <= 1)
            gameObject.SetActive(false);
        else
            gameObject.SetActive(true);
    }
    private GameObject GetFirstChild(GameObject gameObject) =>
gameObject.transform.GetChild(0).gameObject;

}
```

Додаток Д

```
public class PageHistory : MonoBehaviour
{
    public Stack<GameObject> pagesHistory = new Stack<GameObject>();

    public delegate void HistoryHandler();
    public event HistoryHandler OnHistoryChanged;

    private void Start()
    {
        var main = FindObjectOfType<MainMenuController>().gameObject;
        pagesHistory.Push(main);
    }

    public void Add(GameObject gameObject)
    {
        pagesHistory.Push(gameObject);
        OnHistoryChanged.Invoke();
    }
    public GameObject Get() => pagesHistory.Peek();

    public int Count() => pagesHistory.Count;

    public GameObject GoBack()
    {
        GameObject result = pagesHistory.Pop();
        OnHistoryChanged.Invoke();
        return result;
    }
}
```

Додаток Е

```
public class PageSwitcher : MonoBehaviour
{
    private PageHistory pageHistory;

    private void Awake() => pageHistory = FindObjectOfType<PageHistory>();

    public void SwitchToPage()
    {
        GetFirstChild(pageHistory.Get())
            .SetActive(false);

        pageHistory.Add(gameObject);

        GetFirstChild(gameObject).SetActive(true);
    }

    private GameObject GetFirstChild(GameObject gameObject) =>
gameObject.transform.GetChild(0).gameObject;

}
```

Додаток Ж

```
public class CustomNetworkLobbyPlayer : NetworkLobbyPlayer
{
```

```
    public override void OnClientEnterLobby()
    {
        base.OnClientEnterLobby();

        Debug.Log("Client is Enter to Lobby");


        var slots = FindObjectOfType<CustomNetworkLobbyManager>().lobbySlots;
        var playerList = FindObjectOfType<LobbyController>();
        playerList.ShowLobbyPlayer(slots);
    }
}
```