Unity 2D RPG 게임 포트폴리오

FantasyRPG PortFolio

유예원

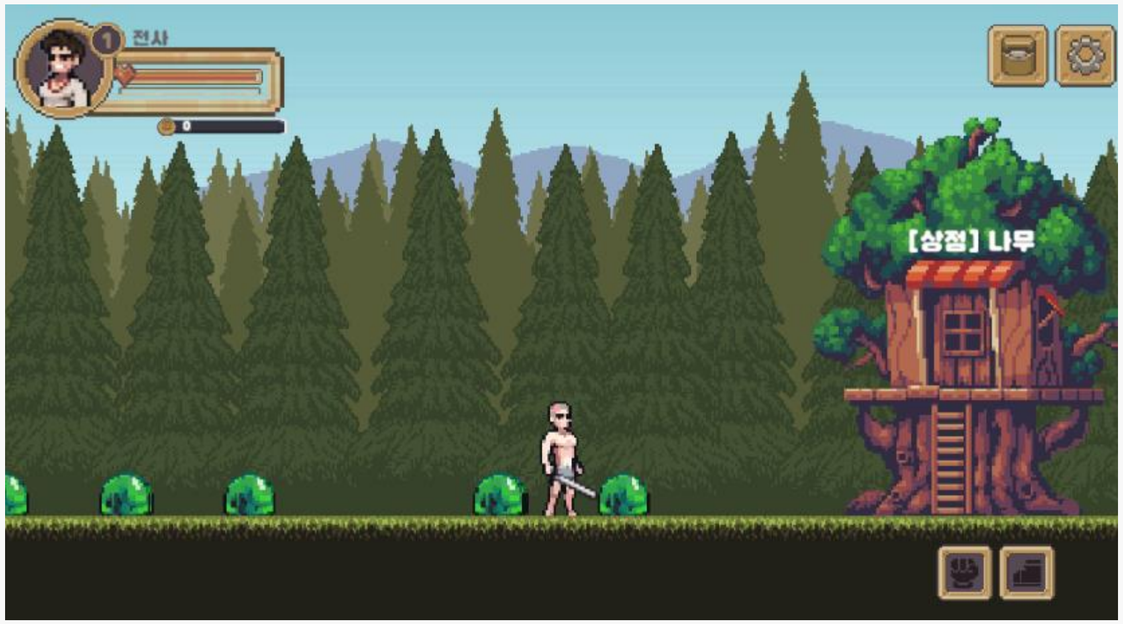
목차

01

프로젝트 개요

어떤 게임인지에 대한 설명 및
깃허브 / 유튜브 링크 첨부

01 프로젝트 개요



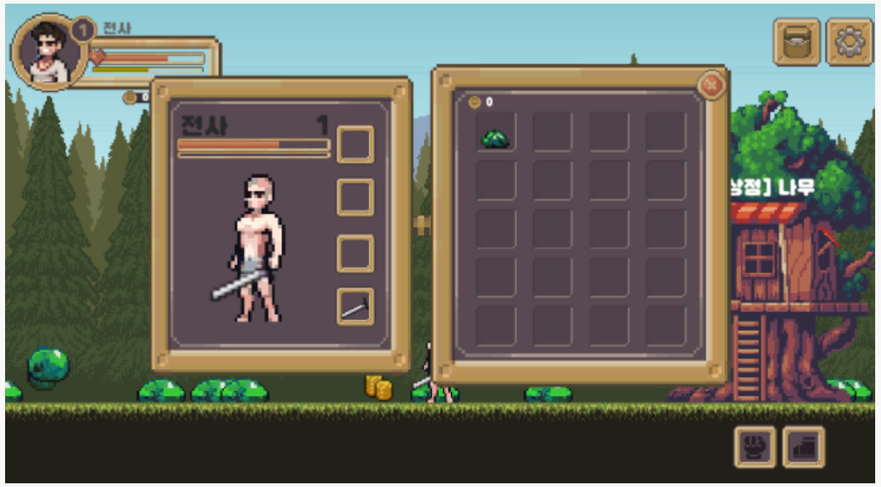
게임 제목 : FantasyRPG
게임 장르 : 횡스크롤 2D 액션 RPG
플랫폼 : 모바일
개발 기간 : 2024.3 ~ 2024.3 (약 2~3주)
개발 환경 : Unity6, C#
주요 시스템 :
- 몬스터 사냥 및 아이템 드랍
- 장비 아이템 수집 및 장착
- 상점과 인벤토리 연동
- 조이스틱 기반 이동

기획 목표	RPG의 기본이라고 할 수 있는 아이템 드랍 및 이를 이용한 인벤토리와 상점을 구현해서 연동해보자
깃허브	https://github.com/xxxuug/FantasyRPG.git
유튜브	https://youtu.be/1c9b0wjhrLw

02

개발 의도

왜 이런 게임을 개발하려고 했나요?



개발 의도

RPG 장르의 기본이라고 할 수 있는 인벤토리 및
상점시스템을 직접 구현해보고 싶었습니다.

특히 모바일 횡스크롤 RPG 게임에서 자주 등장하는 구조를 바탕으로,
익숙하면서도 직관적인 패턴을 따라 구성하려고
노력했습니다.

또한, 성능 최적화와 확장성을 모두 고려하여 설계하였으며,
추후 기능이 추가되더라도 구조적으로 유연하게 확장할 수 있도록 구현했습니다.

03

핵심 기능 구현

어떤 기능들을 어떻게 구현했을까요?

03 핵심 기능 구현 - 플레이어 조작

조이스틱 기반 모바일 조작

UI_JoyStick.cs - 조이스틱 드래그 방향을 GameManager에 전달하여 이동 구현

```
GameManager.Instance.MoveDir = new Vector2(_moveDir.x, 0).normalized;
```

GameManager.cs - 조이스틱 입력 값 받아서 전역 이동 정보로 저장하고 PlayerController에 전달

```
#region JoyStick
public event Action<Vector2> OnMoveDirChanged;

Vector2 _moveDir;

참조 2개
public Vector2 MoveDir
{
    get { return _moveDir; }
    set
    {
        _moveDir = value;
        OnMoveDirChanged?.Invoke(value);
    }
}
#endregion
```

조이스틱 기반의 모바일 조작 시스템과 버튼 입력 방식으로 플레이어의 이동, 점프, 공격, 피격, 사망 등을 처리하였습니다.

애니메이션과 연동된 무기 공격, 피격 시 넉백 및 반투명 처리, 사망 시 애니메이션 정지 등 직관적인 조작 흐름을 구현했습니다.

GameManager를 통해 조작 정보를 통합 관리하고, 버튼과 애니메이션이 일관되게 동작하도록 구성했습니다.

03 핵심 기능 구현 - 몬스터 AI & 드랍 처리

랜덤 이동 + 피격 애니메이션 + 사망 시 아이템 드랍

SlimeGreenController.cs - 아이템 드랍 개수 및 퍼질 각도 설정

```
for (int i = 0; i < itemCount; i++)
{
    float randomAngleOffset = Random.Range(-5f, 5f);
    float baseAngle = (i / (float)itemCount) * 2f - 1f;
    // 아이템 개수에 따라 퍼질 각도 계산
    float angleOffset = baseAngle * spreadAngle + randomAngleOffset;

    GameObject item = dropItems[i];
    item.transform.position = transform.position + new Vector3(0, 1f, 0);

    BaseItem baseItem = item.GetComponent<BaseItem>();
    if (baseItem != null)
    {
        baseItem.SetSpreadAngle(angleOffset);
        baseItem.SetDelay(1.5f);
    }
}
```

BaseItem.cs - 플레이어가 가까이 오면 자동으로 이동해 수거됨

```
if (distance < _playerNearRange)
{
    _collider2D.isTrigger = true;
    _isMovingToPlayer = true;
    _rigidbody2D.linearVelocity = Vector2.zero;
    _rigidbody2D.gravityScale = 0;
}
```

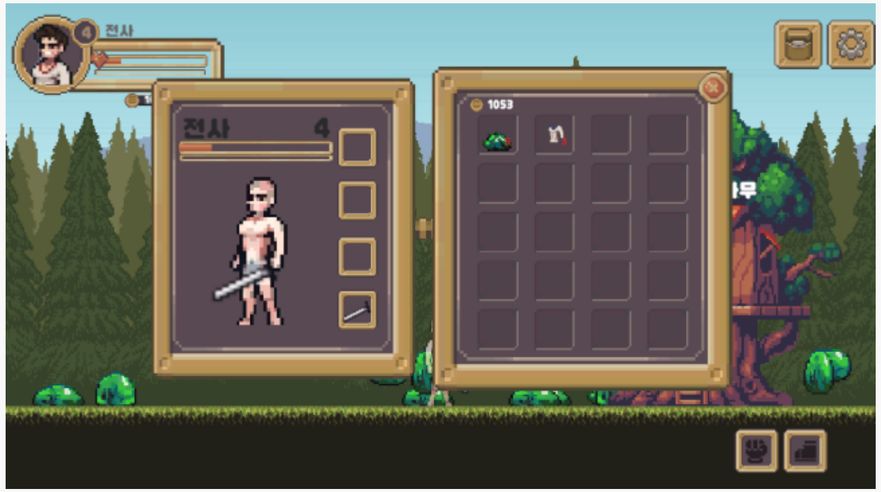
몬스터는 랜덤하게 좌우로 이동하며, 피격 시 애니메이션이 재생되고 일정 시간 정지한 후 다시 이동을 시작합니다.

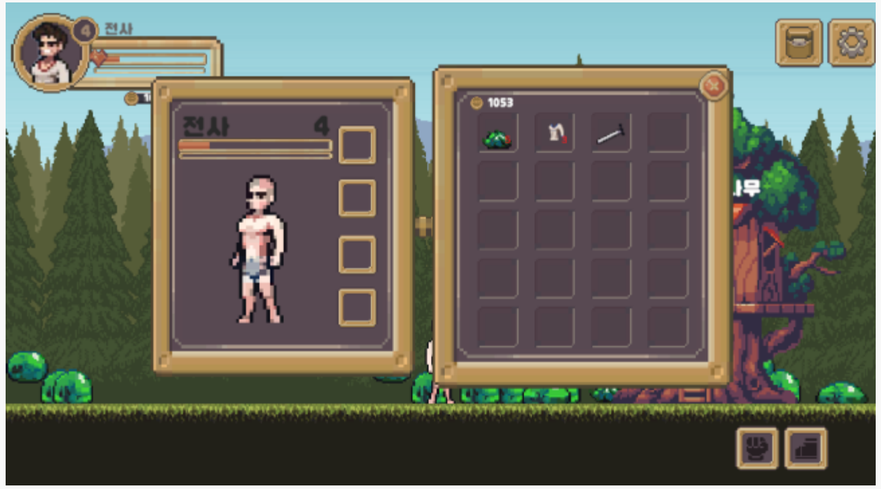
체력이 0이 되면 사망 애니메이션과 함께 골드, 소비 아이템, 장비를 랜덤 확률로 드랍하며, 드랍된 아이템은 일정 시간이 지나면 플레이어를 향해 자동으로 이동하며 획득됩니다.

생성된 아이템은 드랍 각도와 중력을 적용해 자연스럽게 튀어나오는 효과를 표현했고,
모든 오브젝트는 오브젝트 풀링 기반으로 재사용되어 성능 최적화 또한 고려하였습니다.

03 핵심 기능 구현 - 인벤토리 시스템

아이템 저장 / 개수 스택 / 장비 분리 및 장착 시 외형 반영





UI_Inventory.cs - 인벤토리 슬롯에 아이템 추가, 같은 아이템이면 스택

```
public void AddItemToInventory(ItemData item)
{
    Debug.Log("[UI_Inventory] 아이템 추가 시도 : " + item.ItemName);
    foreach (InventorySlot slot in _inventorySlots)
    {
        if (slot.IsSameItem(item))
        {
            slot.IncreaseCount();
            Debug.Log("[U_Inventory] 기존 아이템 개수 증가 : " + item.ItemName);
            return;
        }
    }

    foreach (InventorySlot slot in _inventorySlots)
    {
        if (slot.IsEmpty())
        {
            slot.SetItem(item, 1);
            Debug.Log("[UI_Inventory] 빈 슬롯에 아이템 추가 : " + item.ItemName);
            return;
        }
    }
    Debug.Log("[UI_Inventory] 인벤토리가 가득 참!");
    StartCoroutine(ShowWarningMessage("인벤토리가 가득 찼습니다."));
}
```

InventorySlot.cs - 아이템 슬롯 더블클릭 시 장비 슬롯으로 이동 및 장착 처리

```
// 장비 아이템 장착 해제 함수
참조 1개
public void UnEquip(ItemData item)
{
    foreach (Transform child in transform)
    {
        if (child.name == item.name)
        {
            child.gameObject.SetActive(false);
            break;
        }

    }
}
```

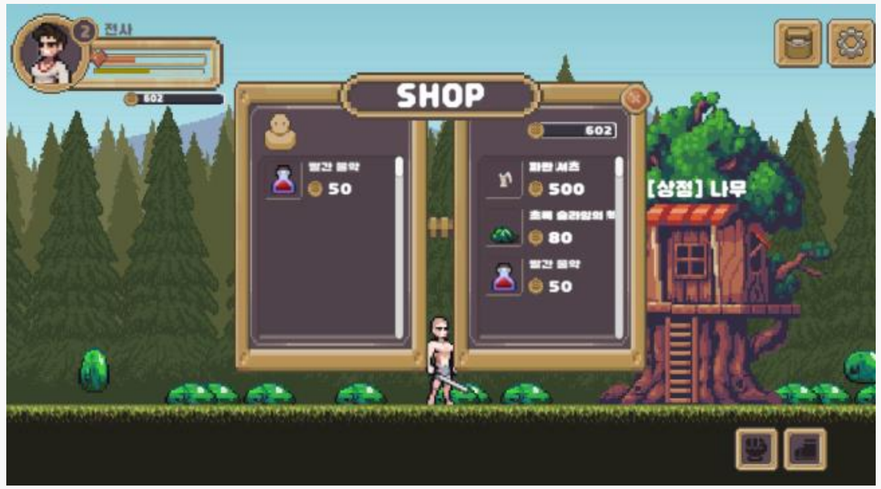
InventoryPlayer.cs - 인벤토리 UI 상에 보이는 캐릭터 외형에도 장비 반영

```
public void OnPointerDown(PointerEventData eventData)
{
    // 같은 슬롯에서 연속 클릭했을 때만
    if (_lastClickedSlot == this && (Time.time - _lastClickTime) < _doubleClickInterval)
    {
        _clickCount++;
        if (_clickCount == 2)
        {
            Debug.Log($"[InventorySlot] {_itemData} 더블클릭");
            _lastClickedSlot = null;
            _lastClickTime = 0;

            // 만약 itemData의 Tag가 Equipment라면
            if (_itemData != null && _itemData.Prefab.CompareTag("Equipment"))
            {
                // 장비 슬롯의 EquipmentType과 itemData의 EquipmentType을 비교해서 같으면 추가
                foreach (EquipmentSlot slot in UI_Inventory.Instance._equipmentSlots)
                {
                    if (slot.SlotType == _itemData.EquipmentType)
                    {
                        // 장비 장착
                        slot.SetItem(_itemData);
                        // 장비 실제 장착 반영...
                        GameManager.Instance.Equip(_itemData);
                        ClearItem();
                        break;
                    }
                }

            }
            return;
        }
    }
```

일반 아이템은 스택 방식으로 저장되고, 장비 아이템은 종류별 슬롯에 구분하여 장착됩니다.
장비 장착 시 플레이어와 인벤토리 캐릭터에 모두 실시간 반영되어 외형이 바뀌며,
같은 장비가 이미 장착되어 있으면 비활성화 후 새 장비로 교체됩니다.
더블 클릭으로 장착/해제가 가능하며, 확장 가능한 구조로 설계하였습니다.

03 핵심 기능 구현 - 상점 시스템

상점 NPC, 아이템 구매/판매 기능



ShopInventorySlot.cs - 인벤토리 아이템 더블클릭 시 판매 팝업 열림

```
public void OnPointerDown(PointerEventData eventData)
{
    // 같은 슬롯에서 연속 클릭했을 때만
    if (_lastClickedSlot == this && (Time.time - _lastClickTime) < _doubleClickInterval)
    {
        _clickCount++;
        if (_clickCount == 2)
        {
            Debug.Log($"[ShopInventorySlot] {_itemData} 더블클릭");
            _lastClickedSlot = null;
            _lastClickTime = 0;

            UI_Shop.Instance.OpenSellPopUp(this, _linkedInventorySlot);
            return;
        }
    }
    else
    {
        _clickCount = 1;
    }
    _lastClickedSlot = this;
    _lastClickTime = Time.time;
}
```

게임 내 상호작용 가능한 상점 NPC를 통해 아이템 구매 및 판매 기능을 구현했습니다.

더블클릭 방식의 UX를 적용해 모바일 환경에서도 직관적인 구매/판매 동작이 가능하게 했습니다.

UI_Shop.cs - 판매확정 시 골드 증가 + 슬롯 수량 감소 or 제거

```
void OnClickSellYES()
{
    SellPopUp.SetActive(false);
    // 인벤토리에서 빠져나가는 로직
    // 개수가 0개면 인벤토리에서 삭제하고 아이템 리스트 위로 정렬
    if (_selectedSellSlot == null)
    {
        Debug.LogError("[UI_Shop] _selectedSellSlot이 null입니다! (판매할 아이템이 선택되지 않았음)");
        return;
    }

    if (_itemData == null)
    {
        Debug.LogError("[UI_Shop] _itemData가 null입니다! (판매할 아이템 정보가 저장되지 않았음)");
        return;
    }

    GameManager.Instance.GetGold(_itemData.Price);

    ShopItemInfo();

    int newCount = _selectedSellSlot.GetItemCount() - 1;

    _selectedSellSlot.SetItem(_itemData, newCount);

    if (newCount <= 0)
        _selectedSellSlot.ClearItem();

    UpdateInventorySlot();
}
```

현재 보유 골드가 실시간 반영되며, 아이템 구매 시 인벤토리에 자동 추가되고 판매 시 수량이 감소하거나 제거됩니다.

상점 시스템은 Inventory 시스템과 연동되어 작동합니다.

04

구조 설계 및 최적화

어떤 식으로 구조를 설계했을까요?

04 구조 설계 및 최적화

전역 구조 설계 : 싱글톤 기반 매니저 시스템

게임 전반의 상태 및 자원 관리를 위해 Singleton<T> 기반 구조를 설계하였습니다.
GameManager, ObjectManager, PoolManager 등 주요 매니저는 전역 접근 가능하도록 구현되어 각 시스템이 상호 유기적으로 연결될 수 있도록 구조화하였습니다.

- Singleton<T>
  - GameManager — 플레이어 상태 및 게임 전체 흐름 관리
  - ObjectManager — 리소스 로드 + 오브젝트 스폰/디스폰 관리
  - PoolManager — 몬스터, 아이템 오브젝트 풀링
  - UI_Inventory — 인벤토리 시스템 UI 전역 접근
  - UI_Shop — 상점 UI 전역 접근

GameManager.cs - 플레이어 정보 및 레벨, 골드, 경험치 등 게임 전체 상태 관리

```
#region PlayerInfo
public event Action OnPlayerInfoChanged;

private PlayerInfo _playerInfo = new PlayerInfo()
{
```

Singleton.cs - 전역 접근 가능한 싱글톤 인스턴스를 생성하고 유지 관리

```
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{
    protected static T _instance = null;
    참조 1개
    public static bool IsInstance => _instance != null;
    참조 0개
    public static T TryGetInstance() => IsInstance ? _instance : null;

    참조 52개
    public static T Instance
    {
        get
        {
            if (_instance == null)
            {
                GameObject manager = GameObject.Find("@Managers");
                if (manager == null)
                {
                    manager = new GameObject("@Managers");
                    DontDestroyOnLoad(manager);
                }
                _instance = FindAnyObjectByType<T>();

                if (_instance == null)
                {
                    GameObject obj = new GameObject(typeof(T).Name);
                    T component = obj.AddComponent<T>();
                    obj.transform.parent = manager.transform;
                    _instance = component;
                }
            }
            return _instance;
        }
    }
```

PoolManager.cs - 몬스터 및 아이템 등 오브젝트들 재사용 가능하게 풀링하여 관리

```
public class PoolManager : Singleton<PoolManager>
{
    Dictionary<System.Type, List<GameObject>> _pooledObject =
        new Dictionary<System.Type, List<GameObject>>();
    Dictionary<System.Type, GameObject> _parentObject =
        new Dictionary<System.Type, GameObject>();

    참조 1개
    public T GetObject<T>(Vector3 pos) where T : BaseController
    {
        System.Type type = typeof(T);

        if (type.Equals(typeof(SlimeGreenController)))
        {
            if (_pooledObject.ContainsKey(type))
            {
                for (int i = 0; i < _pooledObject[type].Count; i++)
                {
                    if (!_pooledObject[type][i].activeSelf)
                    {
                        _pooledObject[type][i].SetActive(true);
                        _pooledObject[type][i].transform.position = pos;
                        return _pooledObject[type][i].GetComponent<T>();
                    }
                }

                var obj = ObjectManager.Instance.Spawn<T>(pos);
                obj.transform.parent = _parentObject[type].transform;
                _pooledObject[type].Add(obj.gameObject);
                return obj;
```

ObjectManager.cs - 리소스 프리팹 로드, 플레이어 및 몬스터 오브젝트 생성 및 관리

```
public T Spawn<T>(Vector3 spawnPos) where T : BaseController
{
    Type type = typeof(T);

    if (type == typeof(PlayerController))
    {
        GameObject obj = Instantiate(_playerResource, spawnPos, Quaternion.identity);
        PlayerController playerController = obj.GetOrAddComponent<PlayerController>();
        _player = playerController;
        Camera.main.GetOrAddComponent<CameraController>();
        return playerController as T;
    }
```

04 구조 설계 및 최적화

성능 최적화를 위한 오브젝트 풀링 & 유틸 구조

반복 생성되는 오브젝트(슬라임, 골드)를 효율적으로 관리하기 위해 오브젝트 풀링(Pooling) 구조를 적용했습니다.
모든 반복 생성되는 오브젝트들은 PoolManager, ObjectManager를 통해 재사용성과 유지보수성, 최적화가 가능합니다.

SpawningPool.cs - 슬라임을 주기적으로 스폰하며 죽은 슬라임은 일정 시간 뒤 재사용

```
void Start()
{
    ObjectManager.Instance.ResourceAllLoad();
    ObjectManager.Instance.Spawn<PlayerController>(new Vector2(0, -1.94f));

    for (int i = 0; i < _maxSpawnCount; i++)
    {
        Vector3 spawnPos = GetRandomPositionOnField(minX, maxX, spacing, occupiedPos);

        SlimeGreenController greenSlime = PoolManager.Instance.GetObject<SlimeGreenController>(spawnPos);
        _spawnSlimes.Add(greenSlime);
    }
}
```

```
IEnumerator CoRespawnMonster()
{
    yield return _spawnInterval;

    foreach (var slime in _deadSlimes)
    {
        Vector3 spawnPos = GetRandomPositionOnField(minX, maxX, spacing, occupiedPos);

        slime.transform.position = spawnPos;
        slime.gameObject.SetActive(true);
    }

    _deadSlimes.Clear();
    _coRespawnPool = null;
}
```

GoldSpawn.cs - 드랍 골드를 오브젝트 풀에서 재사용

```
public GameObject GetGold(Vector3 spawnPos)
{
    for (int i = 0; i < GoldList.Count; i++)
    {
        if (!GoldList[i].activeSelf)
        {
            GoldList[i].SetActive(true);
            GoldList[i].transform.position = spawnPos;
            return GoldList[i];
        }
    }

    GameObject gold = Instantiate(Gold);
    gold.transform.parent = _goldPool.transform;
    gold.transform.position = spawnPos;
    gold.AddComponent<Rigidbody2D>();
    GoldList.Add(gold);
    return gold;
}
```

Extension.cs - 자주 쓰이는 컴포넌트 접근을 간단하게 처리하는 헬퍼 함수

```
public static T GetOrAddComponent<T>(this GameObject go) where T : Component
{
    T component = go.GetComponent<T>();
    if  (component == null)
        component = go.AddComponent<T>();
    return component;
}
```

Define.cs - Animator 해시, 프리팹 경로, 태그 등을 상수로 관리

```
#region Animator
// Player
public readonly static int isMoveHash = Animator.StringToHash("isMove");
public readonly static int KeyboardMoveHash = Animator.StringToHash("KeyboardMove");
public readonly static int JumpHash = Animator.StringToHash("Jump");
public readonly static int GroundHash = Animator.StringToHash("Ground");
public readonly static int AttackHash = Animator.StringToHash("Attack");
```

Unity 2D RPG 게임 포트폴리오

감사합니다.

FantasyRPG 유예원