LUNA ON
포트폴리오

유예원

목차

02

개요

이 게임은?

게임 명 : LUNA ON

제작 기간 : 2025.03 ~ 2025.05 (약 6주)
제작 인원 : 1인
사용 툴 및 기술 : Unity6, C#

게임 장르	3D RPG
기획 목표	퀘스트를 비롯해 보스 몬스터까지 전반적인 3D RPG에서 찾아볼 수 있는 요소들을 가능한 넣어보자
주요 기능	NPC 대화, 퀘스트 수행, 보스몬스터 연출 및 패턴
깃허브	https://github.com/xxxuug/LunaON.git
유튜브	https://youtu.be/WBD_81g0HHI

02

개발 의도

왜 개발하려고 했을까?

현재 제가 만들 수 있는 3D RPG 기능 구현의 한계가 어디인지 궁금했습니다.

조금 부족하더라도 RPG 게임에서 흔하게 만나볼 수 있는 요소들 위주로 구현하려고 노력했습니다.

퀘스트, 보스 몬스터에 중점을 두어 개발했습니다.

03

기능 구현

보스 AI 상태머신 구조

상태 전이 구조, 공격/대기/피격 로직


```
if (context.IsFlying)
{
    Debug.Log($"[DragonIdleState] isFlying == true → TakeOff 진입");
    stateMachine.ChangeState<DragonTakeOffState>();
    return;
}

if (_coolTime < 0f)
{
    if (context.Target && context.IsNearRange())
    {
        if (!context.IsAttacking)
        {
            stateMachine.ChangeState<DragonAttackState>();
        }
    }
    else
    {
        stateMachine.ChangeState<DragonChaseState>();
    }
}
```

```
_stateMachine = new StateMachine<DragonController>(this, new DragonSleepState());
_stateMachine.AddState(new DragonIdleState());
_stateMachine.AddState(new DragonTakeOffState());
_stateMachine.AddState(new DragonFlyFloatState());
_stateMachine.AddState(new DragonDieState());
_stateMachine.AddState(new DragonGetHitState());
_stateMachine.AddState(new DragonChaseState());
_stateMachine.AddState(new DragonAttackState());
```

보스 AI는 유한 상태머신(FSM) 기반으로 공격, 대기, 추적, 피격, 비행 등의 상태를 전환하며 동작합니다.

StateMachine<DragonController>를 사용해 각 상태별 로직을 분리했고, 공격 종료 시점에는 OnAttackAnimationEnd()를 통해 자연스러운 상태 전환을 구현했습니다.

체력이 절반 이하로 떨어질 경우 isFlying 조건을 활성화해 공중 패턴(비행, 메테오 공격 등)으로 전환됩니다.

Hitbox 및 이펙트 처리

claw 공격/flame 공격 등 히트 판정 처리 방식

보스 공격 시 애니메이션 이벤트를 통해 Hitbox가 순간적으로 활성화 되며, 해당 위치에 불꽃, 폭발 등의 이펙트를 재생하여 타격감을 강화했습니다.

이펙트는 전용 함수로 실행되며, 공격 종류마다 다른 시각적 연출을 구현했습니다.

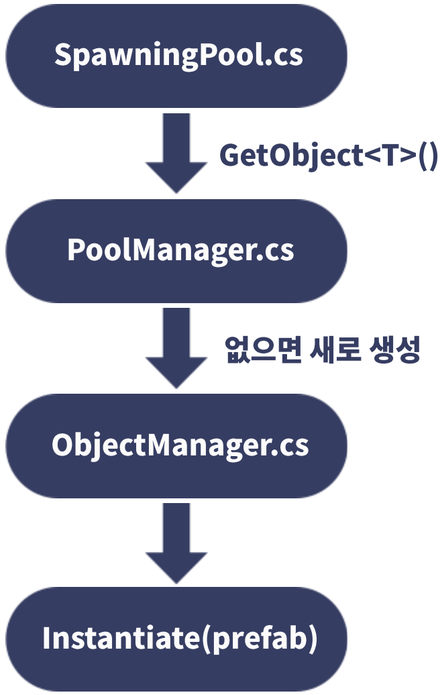
```
void PlayFireEarthEffect()
{
    Instantiate(FireEarthEffect, LeftFireEarthPos.transform.position, LeftFireEarthPos.transform.rotation, LeftFireEarthPos.transform);
    Instantiate(FireEarthEffect, RightFireEarthPos.transform.position, RightFireEarthPos.transform.rotation, RightFireEarthPos.transform);
}
```

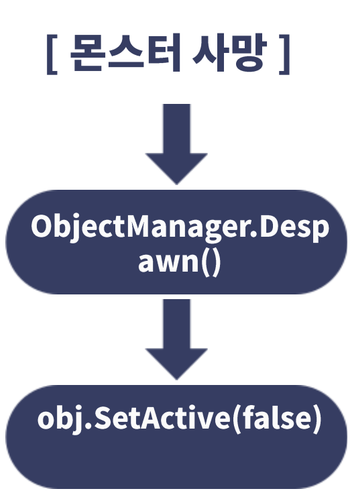
```
참조 1개
void PlayAreaExplosionEffect()
{
    float posZ = Random.Range(25, 89);
    float posX = Random.Range(-175, -138);
    Vector3 randPos = new Vector3(posX, 0, posZ);

    Instantiate(BigExplosionEffect, randPos + new Vector3(0, 1.5f, 0), Quaternion.identity, AreaExplosionPos.transform);
    Instantiate(AreaFireEffect, randPos, Quaternion.identity, AreaExplosionPos.transform);
}
```

```
// Hitbox 이벤트 함수
참조 0개
public void EnableClawHitbox() => ClawHitbox.ActivateHitbox();
참조 0개
public void DisableClawHitbox() => ClawHitbox.DeactivateHitbox();
```

Object / Pool 관리

PoolManager, ObjectManager 구조로 성능 최적화





반복 생성되는 일반 몬스터들은 SpawningPool → PoolManager → ObjectManager 흐름으로 생성/관리됩니다.

PoolManager는 비활성화된 오브젝트를 재사용하고, 부족할 경우 ObjectManager에서 새로 생성합니다.

죽은 몬스터는 SetActive(false)로 풀에 보관되고, 일정시간 후 SpawningPool에서 리스폰 위치에 다시 배치됩니다.

이로 인해 런타임 성능 저하를 방지하고, GC(가비지 컬렉션) 발생을 최소화했습니다.

Object / Pool 관리

PoolManager, ObjectManager 구조로 성능 최적화

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

    if (type.Equals(typeof(CactusController)) || type.Equals(typeof(MushroomController)))
    {
        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;
        }
        else
        {
            if (!_parentObject.ContainsKey(type))
            {
                GameObject go = new GameObject(type.Name);
                _parentObject.Add(type, go);
            }
            var obj = ObjectManager.Instance.Spawn<T>(pos);
            obj.transform.parent = _parentObject[type].transform;
            List<GameObject> newList = new List<GameObject>();
            newList.Add(obj.gameObject);
            _pooledObject.Add(type, newList);
            return obj;
        }
    }
    return null;
}
```

<PoolManager.cs>

<SpawningPool.cs>

```
// 선인장
CactusController cactus = PoolManager.Instance.GetObject<CactusController>(Vector3.zero);
```

<ObjectManager.cs>

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

```
    else if (type == typeof(MushroomController))
    {
        GameObject obj = Instantiate(_mushroomResource, spawnPos, Quaternion.identity);
        MushroomController mushroomController = obj.GetOrAddComponent<MushroomController>();
        Mushrooms.Add(mushroomController);
        return mushroomController as T;
    }
    return null;
}
```

퀘스트 시스템

NPC 거리 인식 + 퀘스트 흐름 설계

QuestManager.cs - 퀘스트 수락

```
public void AcceptQuest(int id)
{
    if (!_activeQuestID.Contains(id))
        _activeQuestID.Add(id);

    OnQuestUpdate?.Invoke();
}
```

UI_Quest.cs - 수락 시 팝업 표시

```
public void ShowQuestNotice(int questId)
{
    QuestUI.SetActive(true);
    QuestManager.Instance.AcceptQuest(questId);
    UpdateQuestUI();
}
```

TalkManager.cs - 수락 분기 대사 처리

```
if (!isAccepted && !isCompleted)
{
    if (quest.StartNPCID == currentNPCID && isRequireLevel)
    {
        SetDialogProgress(_nearNPC, id, 0);
        _currentDialogKey = _nearNPC + "_" + id + "_0";
        StartDialogFromKey(_currentDialogKey);
        return;
    }
}
```

플레이어가 NPC 근처에 접근하면 NPCDistance에서 감지하여 대화 UI의 활성화가 가능해집니다.

대화는 TalkManager에서 키 기반으로 분기되며, 퀘스트 수락 여부에 따라 QuestManager가 처리합니다.
수락된 퀘스트는 UI_Quest에서 팝업 형태로 표시됩니다.

UI_Quest는 OnQuestUpdate 이벤트를 통해 실시칸 퀘스트 정보를 표시하며, 진행도에 따라 자동으로 내용이 갱신됩니다.

구현 포인트 요약

- NPC 거리 인식 → 스페이스 바 클릭 시 대화 노출
- 대사 키 기반으로 수락/진행/완료 분기 처리
- OnQuestUpdate를 통한 UI 연동 갱신
- 타이핑 효과 연출

UI 시스템 설계 구조

공통 Canvas 기반, 모듈형 UI 구성 및 상태 연동 설계

UI_Base

- **UI_Menu** ← 메뉴 패널 열기/닫기 + 애니메이션
- **UI_Boss** ← 보스 버튼 클릭 → 씬 전환 + 포탈 이펙트
- **UI_Quest** ← 퀘스트 수락 시 팝업, 실시간 킬카운트 반영
- **UI_Status** ← 레벨, HP, EXP 실시간 반영
- **UI_Setting** ← 게임 정보, 종료, 설정 패널

UI_Base - 모든 UI의 공통 캔버스 설정

```
protected virtual void Initialize()
{
    SetCanvas();
}

참조 1개
private void SetCanvas()
{
    Canvas canvas = gameObject.GetOrAddComponent<Canvas>();
    if (canvas != null)
```

UI_Status - 실시간 UI 반영

```
SceneManager.sceneLoaded += OnSceneLoaded;
GameManager.Instance.OnPlayerInfoChanged += UIUpdate;
UIUpdate();
```

UI_Boss - 보스 버튼 → 씬 전환

```
void OnClickBossEnter()
{
    BossUI.SetActive(false);
    ShowPotalEffect();
    StartCoroutine(EnterBossRooom());
}
```

UI_Menu - 슬라이딩 애니메이션 처리

```
IEnumerator SlideMenuPanel(float from, float to)
{
    float elapsed = 0; // 경과시간

    while (elapsed <= _duration)
    {
        float t = elapsed / _duration;
        float width = Mathf.Lerp(from, to, t);

        Vector2 size = _menuPanelRectTransform.sizeDelta;
        size.x = width;
        _menuPanelRectTransform.sizeDelta = size;

        elapsed += Time.deltaTime;
        yield return null;
    }

    Vector2 finalSize = _menuPanelRectTransform.sizeDelta;
    finalSize.x = to;
    _menuPanelRectTransform.sizeDelta = finalSize;

    if (to == 0)
        TotalMenu.SetActive(false);
}
```

구현 포인트 요약

- 모든 UI는 UI_Base 상속 → 공통 해상도/Canvas 설정 통일
- 상태 변경 시 이벤트(OnQuestUpdate, OnPlayerInfoChanged) 기반 UI 실시간 반영
- UI_Menu는 클릭 영역 분리 및 슬라이딩 애니메이션 처리
- UI_Boss는 클릭 → 효과 출력 → 씬 전환까지 통합 설계

Player 조작 및 무기 시스템 설계

마우스 조작 + 무기 히트박스 구현

PlayerController는 이동/점프/공격/피격/사망 등 모든 액션을 제어하며,
공격 시 WeaponHitbox가 활성화되어 적의 IDamageable 인터페이스로 피해를 전달합니다.
애니메이션 상태는 PlayerStateMachineBehavior로 관리되어, 콤보 공격 흐름과 무기 이펙트 타이밍이 정교하게 동기화됩니다.

WeaponHitbox.cs – 검 히트박스 구현

```
private void OnTriggerEnter(Collider other)
{
    IDamageable damageable = other.GetComponent<IDamageable>();
    if (damageable == null || damageable.Tag != Define.EnemyTag) return;

    if (other.gameObject.CompareTag("Enemy") && _canHit)
    {
        float atk = GameManager.Instance.PlayerInfo.Atk;
        // IDamageable에 검 자체가 아닌 플레이어를 넘겨야 몬스터가 플레이어를 추격할 수 있음
        GameObject player = ObjectManager.Instance.Player.gameObject;
        damageable.AnyDamage(atk, player);
        DisableAttack();
    }
}
```

PlayerStateMachineBehavior.cs – 플레이어 콤보 공격 상태머신 구현

```
public override void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    base.OnStateUpdate(animator, stateInfo, layerIndex);

    float currentTime = stateInfo.normalizedTime;
    bool isNextCombo = animator.GetBool(Define.isNextCombo);
    bool isAttacking = animator.GetBool(Define.isAttacking);

    if (currentTime < 0.98f && currentTime > 0.25f && isNextCombo)
    {
        // 전투 대기시간 초기화
        PlayerController._battleIdleTime = 0f;
        int comboCount = animator.GetInteger(Define.ComboCount);
        comboCount = comboCount < 3 ? ++comboCount : 0;
        animator.SetInteger(Define.ComboCount, comboCount);

        //Debug.Log($"[SMB] 콤보 카운트 증가 → {comboCount}");

    }
```

공통 유틸리티 / 싱글톤 설계 구조

최적화를 위한 구조 구현

Singleton.cs

```
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;
            }
            else
            {
                _instance.transform.parent = manager.transform;
            }
        }
        return _instance;
    }
}
```

Define.cs

```
#region Input
public const string MouseX = "Mouse X";
public const string MouseY = "Mouse Y";
public const string MouseScroll = "Mouse ScrollWheel";
public const string Horizontal = "Horizontal";
public const string Vertical = "Vertical";
#endregion

#region Animation
public readonly static int Speed = Animator.StringToHash("Speed");
public readonly static int ComboCount = Animator.StringToHash("ComboCount");
public readonly static int isNextCombo = Animator.StringToHash("isNextCombo");
```

**ObjectManager, GameManager, QuestManager, PoolManager 등 주요 매니저들은 공통 싱글톤 기반으로 관리되며,
Define.cs 에는 프로젝트 전역에서 사용되는 상수 및 태그를 통합 관리합니다.
이 구조는 반복 사용되는 기능을 일관되게 관리하고 유지보수성을 향상시키기 위해 설계되었습니다.**

구현 포인트 요약

- Singleton 패턴을 통해 전역 접근성 보장 + 중복 생성 방지
- Deinfe.cs로 상수값/애니메이터 파라미터 관리 → 오타 방지 및 유지보수 용이
- 프로젝트 확장 시, 새로운 매니저도 같은 방식으로 간편하게 추가 가능
- 인스턴스 호출 명확하게 일관화 가능

THANK YOU

게임 개발자 유예원
LUNA ON 포트폴리오