

이태준
Game Developer
포트폴리오



이태준

TAEJUNE LEE

2001.02.28

ltjbs2020@gmail.com

010-2414-5526

내가 작성한 코드는 언제든지 상황에 따라서 수정될 수 있다고 생각합니다.
항상 추가/수정이 용이한 코드를 작성할 수 있도록 노력합니다.



학력

2016.03 ~ 2019.02 한세사이버보안고등학교 게임과 졸업

보유기술

Unity3D
WPF
DirectX

경력

2019.01 ~ 2019.07 디지털존 응용프로그램 개발
KIOSK 대학증명서 발급 서비스 개발
전통문화사업 연구개발 프로젝트 : 전형지원 프로그램 개발
풋볼팬타지움 : 가상 유니폼 착용 서비스 개발

자격증

ITQ 한글, 엑셀 1급
정보처리기능사

Contents

1 프로젝트 개요

2 Scene 구성

3 콘텐츠 구현

- 데이터 처리
- 무기
- 몬스터
- 퀘스트

4 최적화

AssetBundle
Coroutine/Task
ObjectPooling
해상도 대응

5 소감

프로젝트 개요

게임 명 : ProjectPocket
플랫폼 : Mobile/Android
장르 : RPG/어드벤처

개발 스킬 : Unity3D/C#/ MSSQL

핵심 기술 :
ObjectPooling
MSSQL DB 서버, JSON 파싱
Coroutine/Task 비동기 처리
디자인패턴

개발 인원 : 1인 개발

개발 기간 : 2020.03 ~ 2020.06 (약 3개월)

소개

게임개발을 공부하기 시작한 순간부터 RPG 게임을 개발해보는 것이 꿈이었습니다.

최근에 구글 플레이 스토어에 출시된 게임인 “포켓월드 : 탐험의 섬” 라는 게임을 접하게 되었는데

저의 취향에 맞았고 굉장히 재밌게 플레이 하여 카피캣 프로젝트로 개발해보고 싶다는 생각을

하게 되어 RPG 게임을 개발하겠다는 꿈도 실현시키고자 개발하게 된 게임입니다.

전투/채집/퀘스트 가 주요 콘텐츠인 게임으로, 거시적인 목표나 엔딩없이 탐험하는 것이

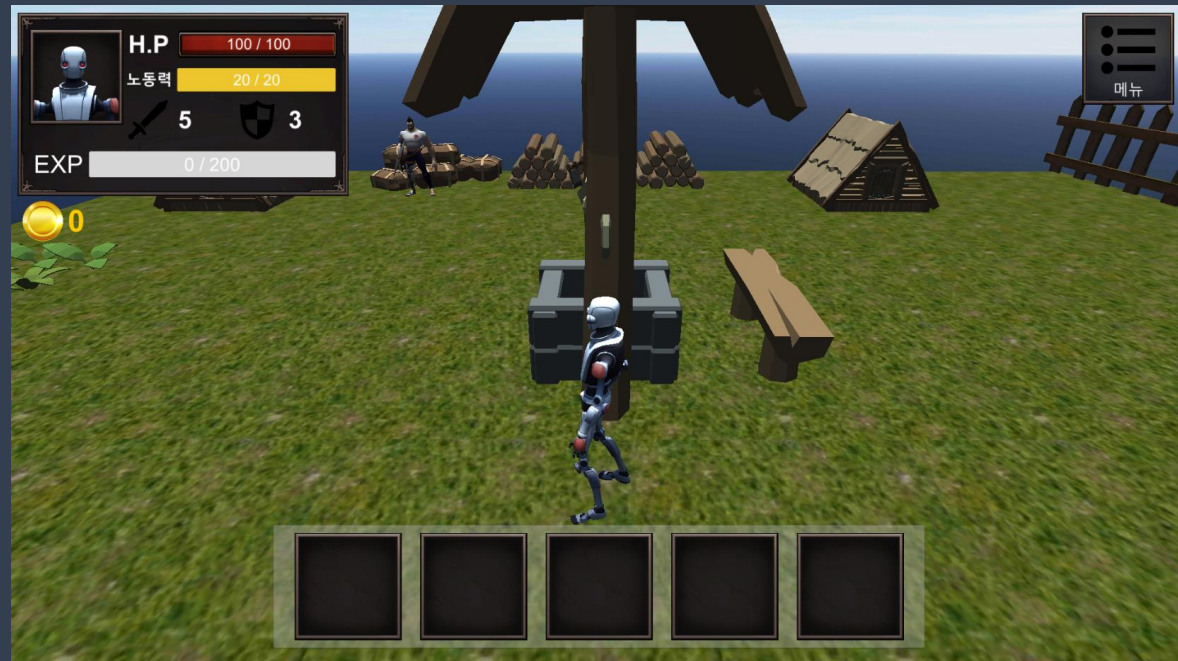
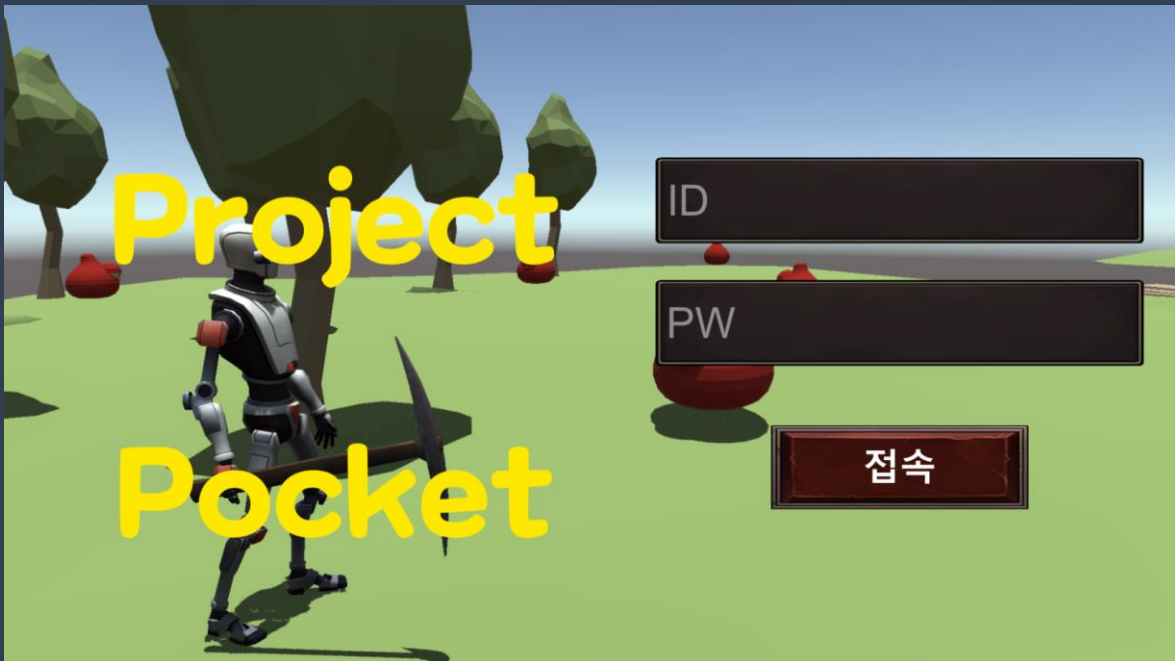
목표인 RPG/어드벤처 게임 입니다.



Github

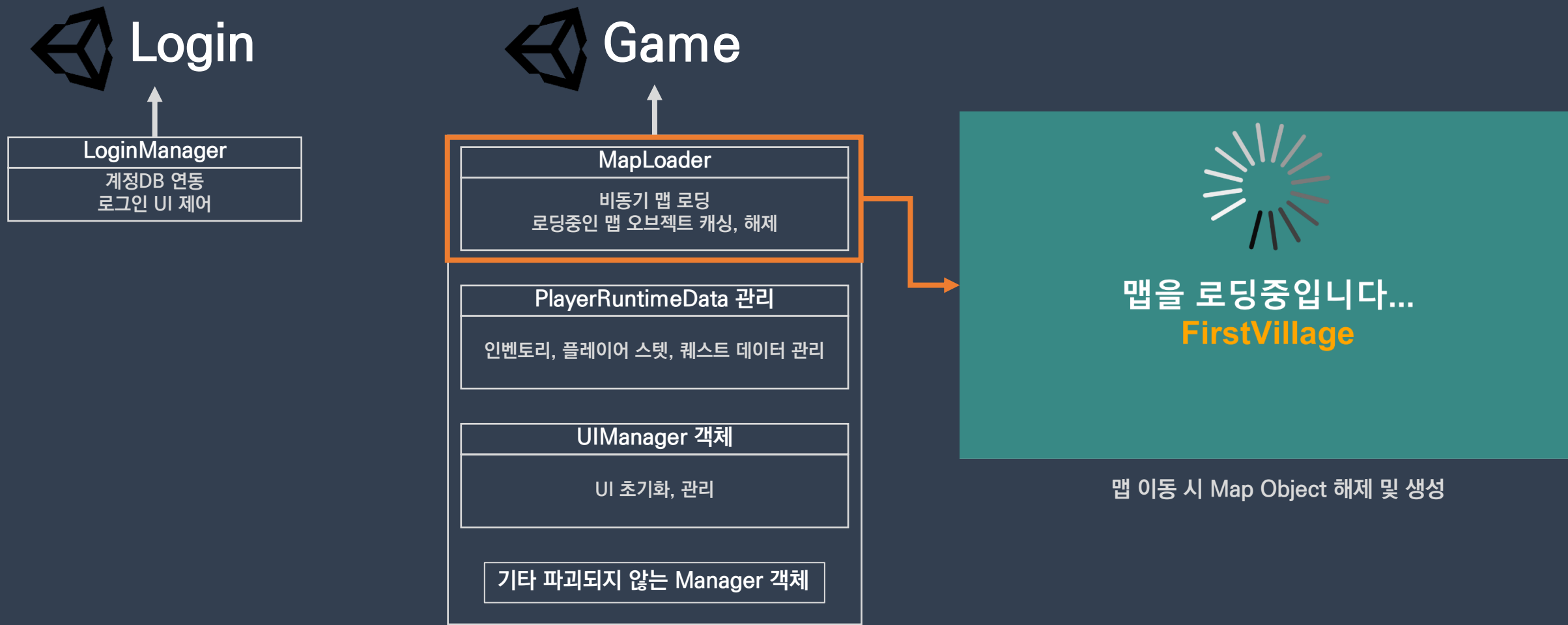


영상



Scene 구성

계정 DB 연동, 로그인 기능을 담당하는 Login Scene
게임 주요 시스템 구현, 맵 생성/해제 등의 기능을 담당하는 Game Scene
총 2가지 Scene으로 구성하여 개발하였습니다.



컨텐츠 : 데이터 처리

게임 내의 모든 데이터를 MSSQL DB서버를 사용하여 관리하도록 개발하였습니다.



Database 서버

DB 이름	내용
Account_DB	플레이어 계정 정보 DB
Game_DB	아이템, 퀘스트, NPC, 몬스터 등 게임 내부 데이터 DB
Item_DB	장비, 소모품 등 아이템 DB
PlayerInfo_DB	능력치, 로그인 맵, 플레이어 정보 DB



DBConnector.cs

DB서버 <-> 런타임 데이터
중간 관리 역할 수행
DB 접속 및 DML 쿼리 API 제공



런타임 데이터 관리

DB 서버통신 원본데이터 캐싱 및 저장	실행 중 플레이어 데이터 캐싱, 관리	실행 중 게임내부 데이터 캐싱, 관리
UserInfoProvider UserInventoryProvider UserEquipmentProvider UserQuestProvider UserQuickSlotProvider UserBuildingProvider	PlayerStat PlayerInventory PlayerEquipment PlayerQuest PlayerQuickSlot PlayerBuilding	ItemDB NpcDB QuestDB MonsterDB BuildingDB ResourceDB

런타임 중 데이터를 관리할 객체들을
Singleton으로 구현하였습니다.

데이터 저장 요청



PlayerDataSaver.cs

런타임 데이터 저장
Singleton Class

플레이어 데이터 UPDATE 실행

컨텐츠 : 데이터 처리

가변 길이 데이터(인벤토리, 퀘스트 등) 과 같이 길이가 일정하지 않은 데이터를
처리하고 데이터 변경 적용을 빠르게 하기 위하여 일부 Record 에서 JSON 파싱을 사용하였습니다.



dbo.PlayerInventory

UserAccount	ItemSlot_0	ItemSlot_1	ItemSlot_2	ItemSlot_3
Account_0	10001	10002	20001	30001
Account_1	40012	40001	10003	10001
Account_2	20003	20002	10004	10001
Account_3	40001	10002	10003	20004

10/10/16

인벤토리 테이블 Colum을 일반 자료형으로 할 경우
인벤토리 크기가 고정 되거나 가변적으로 할 수 없는 문제 발생

UserAccount	InventoryJSON
admin	{"ItemUnits":[{"ItemCode":10003,"ItemCount":1},
test	{"ItemUnits":[{"ItemCode":10001,"ItemCount":1},
test1	{"ItemUnits":[]}
NULL	NULL

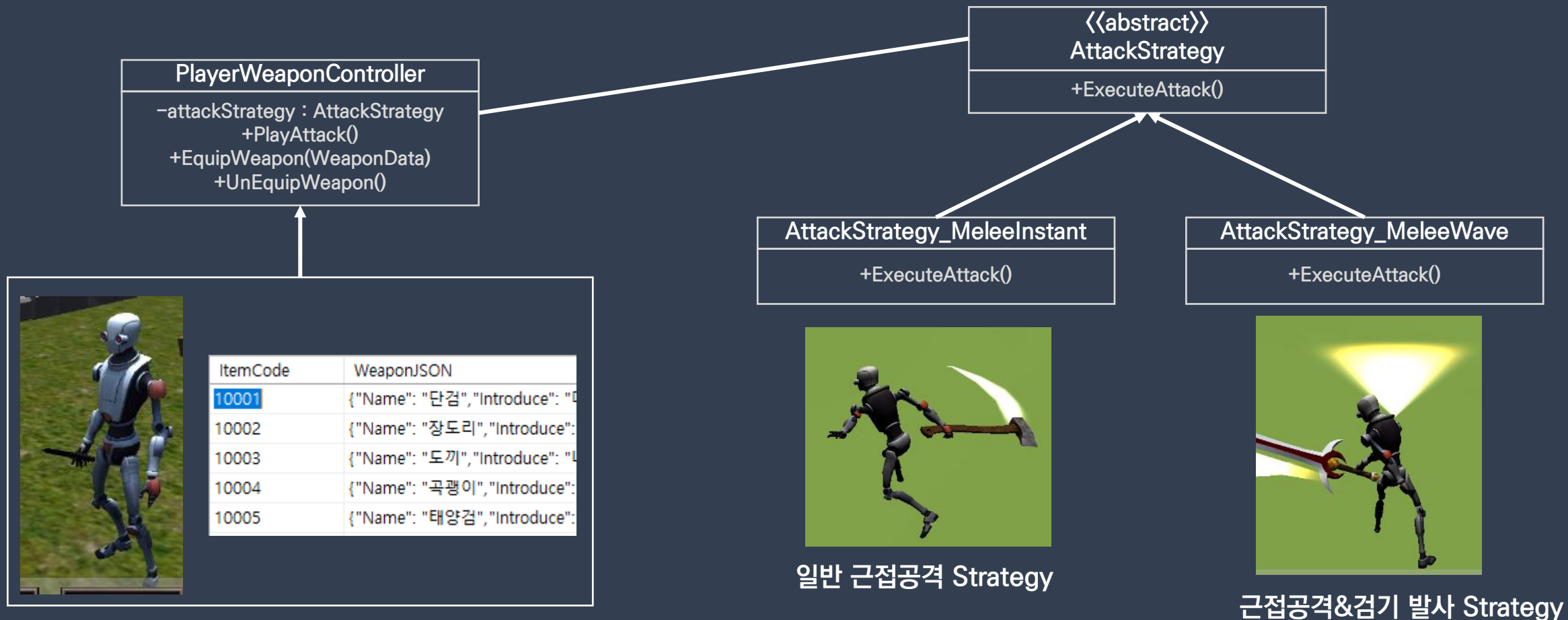

```
[System.Serializable]
public class InventoryJSON
{
    public InventoryJSONUnit[] ItemUnits;
}

[System.Serializable]
public class InventoryJSONUnit
{
    public int ItemCode;
    public int ItemCount;
}
```

JSON 파싱을 사용하여
가변길이 데이터를 관리할 수 있게 되었고,
간편한 직렬화/비직렬화 시스템을 통하여
빠르게 수정한 데이터를 테스트할 수 있었습니다.

컨텐츠 : 무기

무기별로 다른 동작을 구현하기 위해서 **Strategy** 패턴을 사용하여 개발하였습니다.
무기의 공격기능을 공격전략(AttackStrategy)으로 캡슐화 하여 구현하였습니다.

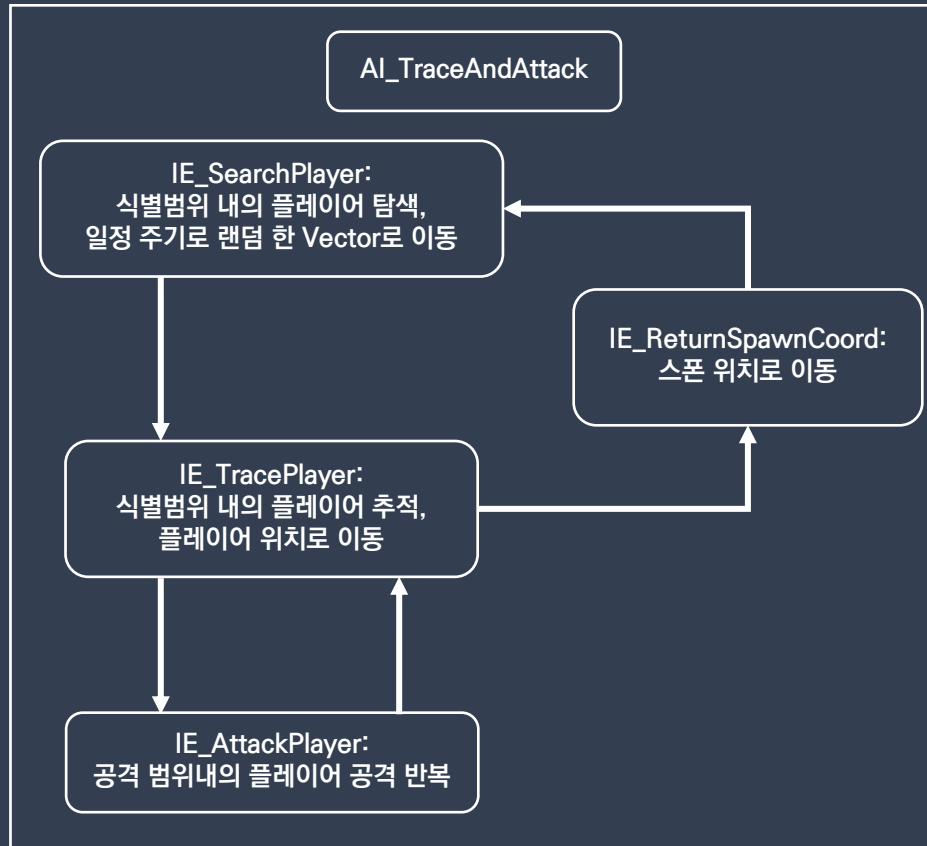


무기 DB 데이터를 토대로 무기 모델을 불러옵니다.

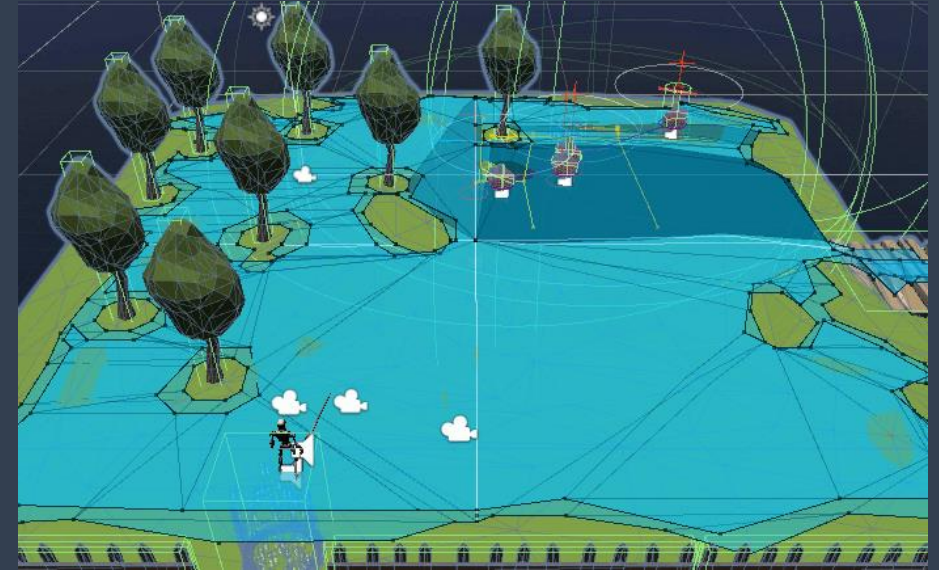
컨텐츠 : 몬스터 AI

FSM(유한상태머신) 구조를 사용하여 몬스터의 AI를 구현하였습니다.

추적&공격 AI 실행도



AI 행동단위 구현을
Coroutine을 사용하여 구현하였습니다.

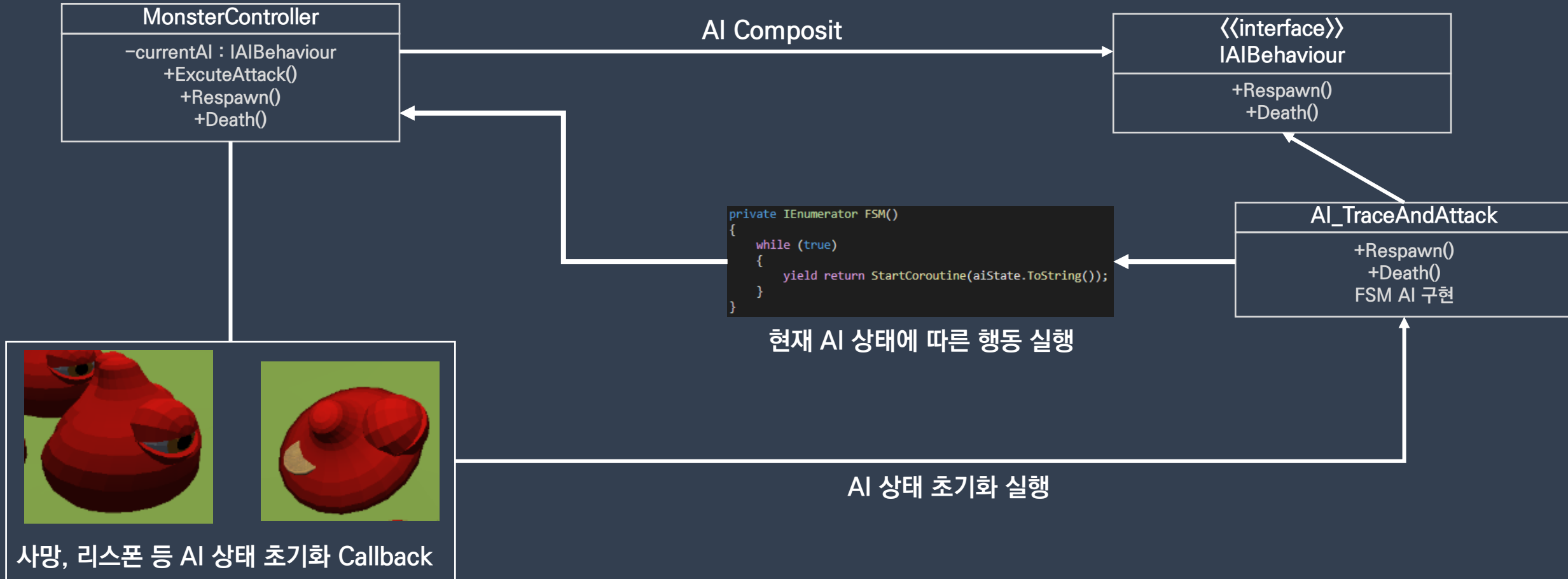


Nav Mesh, Agent 를 사용하여
AI의 이동을 구현하였습니다.

```
NavAgent.speed = Stat.MoveSpeed;  
NavAgent.SetDestination(newDestination);  
...
```

컨텐츠 : 몬스터 AI

몬스터 AI 기능을 캡슐화 하여 interface 로 분리하였습니다.
몬스터 제어 객체는 AI Interface를 구현하는 객체를 구성(Composit) 하도록 구현하였습니다.
그로 인해서 몬스터 AI 변경/추가 시에도 몬스터 제어 객체의 코드 수정이 없도록 개발하였습니다.



컨텐츠 : 퀘스트

몬스터 사냥, 건물 업그레이드, 아이템 획득, NPC 대화 총 4가지 종류의 퀘스트를 개발하였습니다.



아이템 획득, 건물 업그레이드 퀘스트의 경우
각각 플레이어 인벤토리, 건물 데이터 조회로 진행상황을 확인할 수 있음



NPC 대화, 몬스터 사냥 퀘스트는 퀘스트를 진행 중 인 경우에만
진행상황을 조회할 수 있으므로 별도의 진행상황 **Container Class**를 구현하여 개발하였습니다.

컨텐츠 : 퀘스트

퀘스트 진행상황 알림 팝업을 구현하기 위하여 **Observer** 패턴을 적용하여 개발하였습니다.



최적화 : AssetBundle

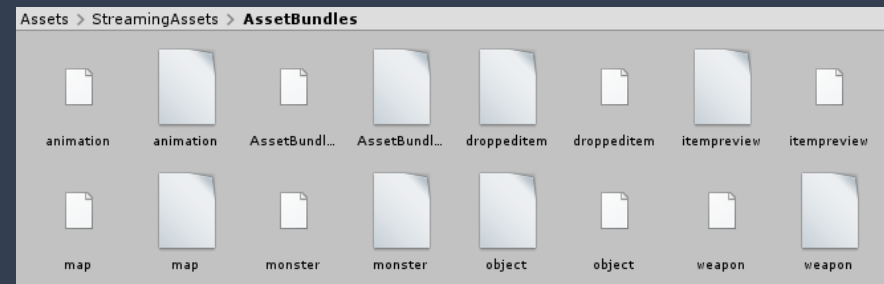
게임전체에 동적 Object 생성 방식을 **Resources** 폴더를 사용하여 구현하였었습니다.
하지만 Resources.Load 방식의 근본적인 **로딩 속도 이슈**가 발생하였습니다
AssetBundle을 사용하는 방식으로 변경하여 해결하였습니다.



맵을 로딩중입니다...
FirstVillage

Resource.Load 방식은
속도가 느려, UI스레드가 멈추는 현상 발생

C# Native 파일 로드,
Unity DOTS 등 시도하였으나
지원하지 않거나
Resources 방식의 근본적인 한계로
실패



```
public class AssetBundleCacher : MonoBehaviour
{
    [Singleton]
    // Data
    private Dictionary<string, AssetBundle> bundles;

    public bool HasAlreadyCachingBundle(string name)...
    public Object LoadAndGetAsset(string bundleName, string assetName)...
    public AssetBundle LoadAndGetBundle(string bundleName)...
    public void CachingBundle(AssetBundle bundle, string name)...
    public AssetBundle GetBundle(string name)...
}
```

AssetBundle 캐싱 관리 Manager

기존 Resource.Load 에서
AssetBundle 방식으로 변경하여 해결

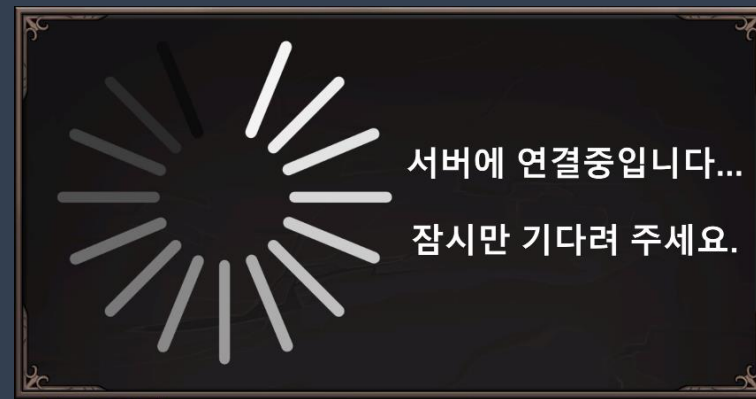
최적화 : Coroutine/Task

플레이어 데이터 저장, 로그인, 맵 로딩 과 같은 시간이 오래 걸리는 작업들을
Coroutine, C# Task 를 사용하여 비동기로 처리함으로써 로딩화면을 구현하였습니다.



```
if (AssetBundleCacher.Instance.HasAlreadyCachingBundle(bundleName))
    bundle = AssetBundleCacher.Instance.GetBundle(bundleName);
else
{
    var request = AssetBundle.LoadFromFileAsync($"{Application.streamingAssetsPath}/AssetBundles/{bundleName}");
    yield return request;
    bundle = request.assetBundle;
    AssetBundleCacher.Instance.CachingBundle(bundle, bundleName);
}
var assetRequest = bundle.LoadAssetAsync<GameObject>($"Map_{loadedMapName}");
yield return assetRequest;
```

맵 로딩과 같이 용량이 큰 동적 Object 로드 시에
AssetBundle.LoadAsync 를 활용하여 비동기로 로드 하였습니다.



```
ShowLoadingPopup();
Task<string> accountTask = Task<string>.Factory.StartNew(
    () => DBConnector.Instance.ValidateAccountOnDB(id, pw));
await accountTask;
string accountResult = accountTask.Result;
```

로그인 과 같은 DB 연동시에 오래 걸리는 작업 들에서
C# Task를 사용하여 비동기로 실행하였습니다.

최적화 : ObjectPooling

자주 On/Off 되는 Object들을 **ObjectPooling** 기법을 사용하여 메모리 할당/해제 가 최소화 되도록 개발하였습니다.



예) 몬스터 리스폰 구현



예) 검기 발사 무기구현

```
private void CreateMonsterPool()
{
    for (int i = 0; i < MaxSpawnCount; ++i)
    {
        GameObject newMonster = Instantiate(SpawnMobPrefab, transform);
        newMonster.GetComponent<MonsterController>().Initialize(DeathMonster);
        newMonster.gameObject.SetActive(false);

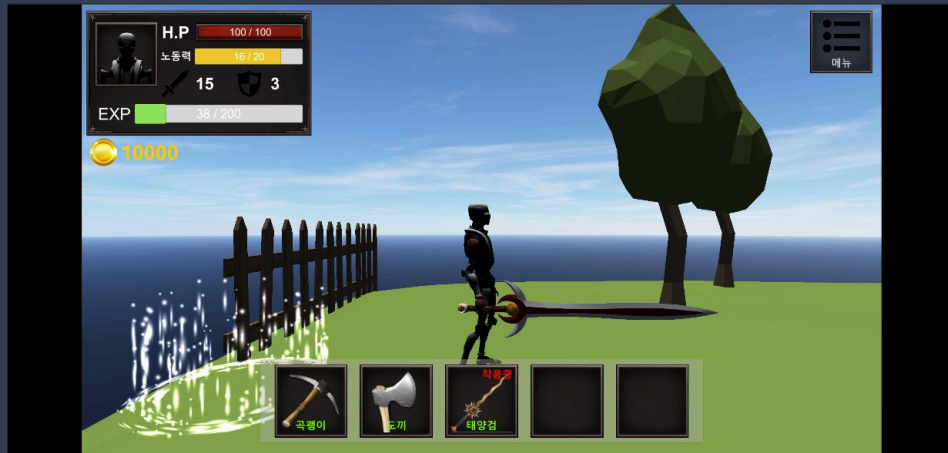
        deactiveMobPool.Add(newMonster);
    }
}
```

미리 정해진 최대치만큼 몬스터를 생성하여 Respawn 하도록 개발하였습니다.

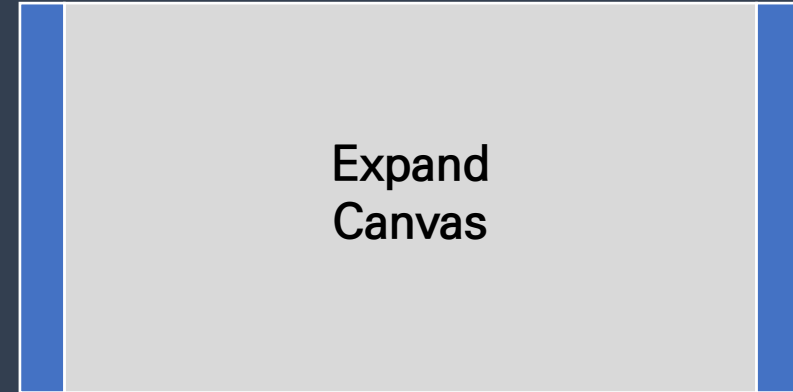
```
private void CreateWavePool()
{
    GameObject projectilePrefab = AssetBundleCacher.Instance.LoadAndGetAsset("weapon", "WaveProjectileBox") as GameObject;
    for (int i = 0; i < 10; ++i)
    {
        ProjectileColiderBox newProjectile = Instantiate(projectilePrefab).GetComponent<ProjectileColiderBox>();
        newProjectile.transform.parent = transform;
        newProjectile.Initialize(projectileTrailColor, ReturnToPool);
        deactiveProjectilePool.Add(newProjectile);
    }
}
```

비활성화, 활성화 의 두가지 검기 Pool을 생성하여 비활성화 상태의 Object가 부족할 시 Pool의 크기를 늘리지 않고 활성화된 Object를 비활성화 시켜서 재활용하는 방식으로 구현하였습니다.

최적화 : 해상도 대응



기본적으로 16:9 Landscape 해상도를 지원하도록 개발하였습니다.
만약 핸드폰의 종횡비가 16:9와 맞지 않는다면
레터박스를 생성하도록 개발하였습니다.



또한 최신 핸드폰 기종들이 가로 회전 시 종횡비가 16:9 보다 가로가 길어지는 것을 고려하여 UI Canvas를 Expand로 설정하여 UI 레이아웃이 변형되지않도록 구현하였습니다.

- 데이터 관리 부분에서 처음으로 DB 서버를 사용하게 된 프로젝트입니다. 자잘한 이슈가 있었지만, 본격적으로 많은 량의 데이터를 분류하고(아이템 코드, 퀘스트 코드 등) 적용해보면서 데이터 관리를 설계하는 데에 있어서 한 단계 성장하게 되었다고 생각합니다.
- 로직 설계 부분에서 Singleton 정도를 제외하면 디자인 패턴을 적용하는 데에 어려움이 있었는데, 프로젝트를 진행하면서 설계 단계에서 캡슐화를 고려하며 진행하여 행위(Behaviour) 종류의 디자인 패턴을 본격적으로 사용해볼 수 있었고, 문제해결능력 이 많이 상승하였다고 느끼게 되었습니다.
- 사람 이므로 완전무결한 코드를 작성할 수는 없지만, 이번 프로젝트를 진행하면서 시도해본 것처럼 여러 기술, 기법들을 활용하며 더욱 발전하여 완벽에 가깝게 코드를 작성할 수 있는 개발자가 되고 싶습니다.

감사합니다