유니티 포트폴리오

작성자 - 심교남

개발 도구 - Unity Engine(2019.2.2f1), Visual Studio 2017

작성일자 - Dec. 2019

Cleaning Ball

안드로이드 기반의 아케이드 게임입니다. 사냥을 하고 성장하는 간단한 게임입니다.

0. 게임 설명

제목 - Cleaning Ball

장르 - 캐주얼, 아케이드

플랫폼 - 안드로이드

플레이 영상 - https://blog.naver.com/tlaryska12/221694602699

0. 게임 설명

플레이어를 가상 조이스틱으로 조작할 수 있습니다.
플레이어는 돌아다니면서, 터렛 들을 몰고 다니고,
터렛 들을 통해, 나쁜 적들을 섬멸하는 게임입니다.

적을 처치하면 재화를 획득할 수 있습니다.
이 재화는 플레이어의 강화에 이용할 수 있습니다.
난이도를 올려, 획득 재화를 늘릴 수도 있습니다.

적들은 플레이어가 다가오면 쫓아오며 공격합니다.
적들에게 당하면 플레이어의 체력 바가 줄어듭니다.

체력 바가 다 줄면, 모든 재화를 잃게 됩니다.

0. 게임 설명

획득한 재화를 통해 강화를 할 수 있습니다.
또한 따라다니는 터렛의 배치를 바꿀 수 있습니다.
데이터를 초기화하고, 다시 시작할 수도 있습니다.

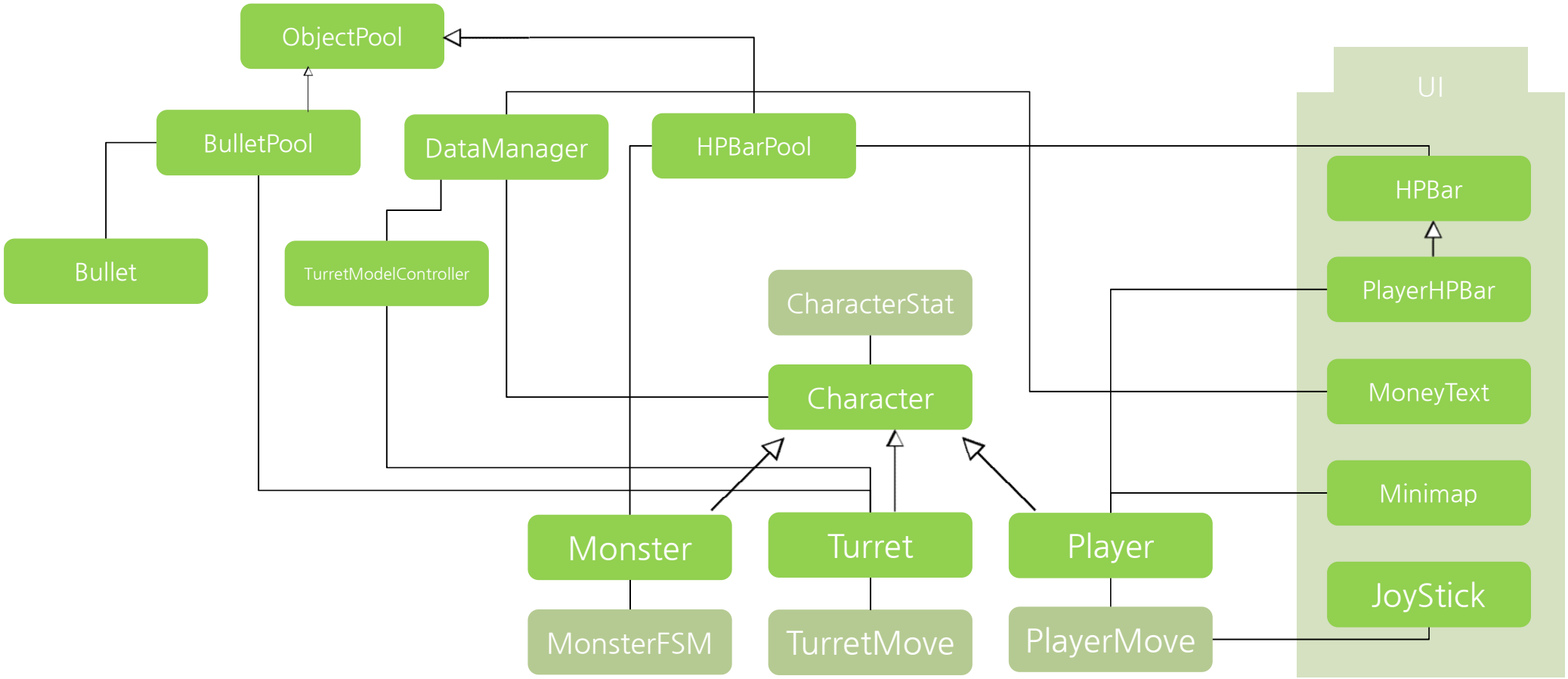
1. 개발 기간

개발 기간 : 3주
하루 2시간에서 5시간씩 작업하였습니다.
1인 개발을 하다 보니 불가피하게 모델은 에셋스토어의 에셋을 사용하였고,
UI는 파워포인트로 직접 디자인 하였습니다.
사운드도 인터넷에서 구하였습니다.

주	내용
1주차	플레이어의 이동과 몬스터 FSM, 몬스터 랜덤 생성 구현
2주차	플레이어 공격 구현, 프로토타입 완성 이후 상점 구현,
3주차	사운드 삽입, 게임 플레이하며 버그 찾기.

2. 설계 - 클래스 구조도 - 전투

2. 설계 - 클래스 구조도 - 전투

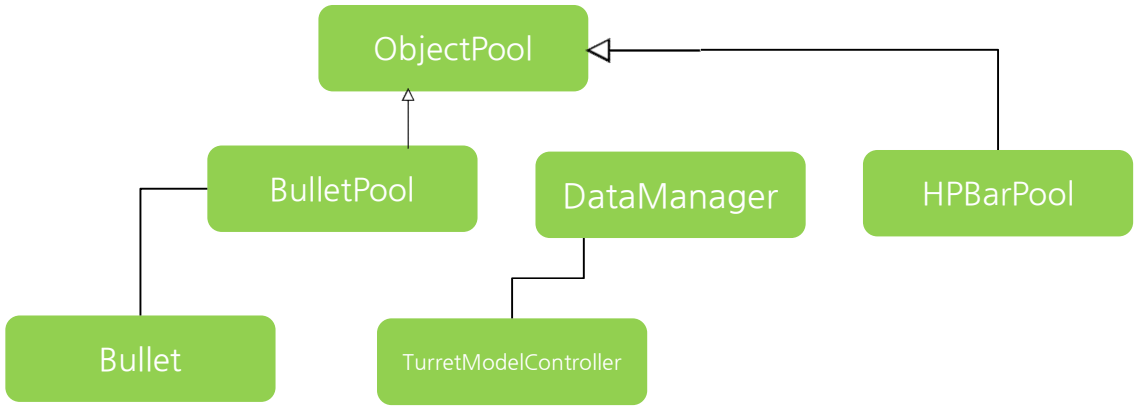


2. 설계 - 클래스 구조도 - 전투

UI
화면에 표시되는 정보들을 담당하는 클래스 입니다.
가상 조이스틱, 미니맵, 체력바 를 구현합니다.
모두 UGUI를 통해서 개발되었습니다.

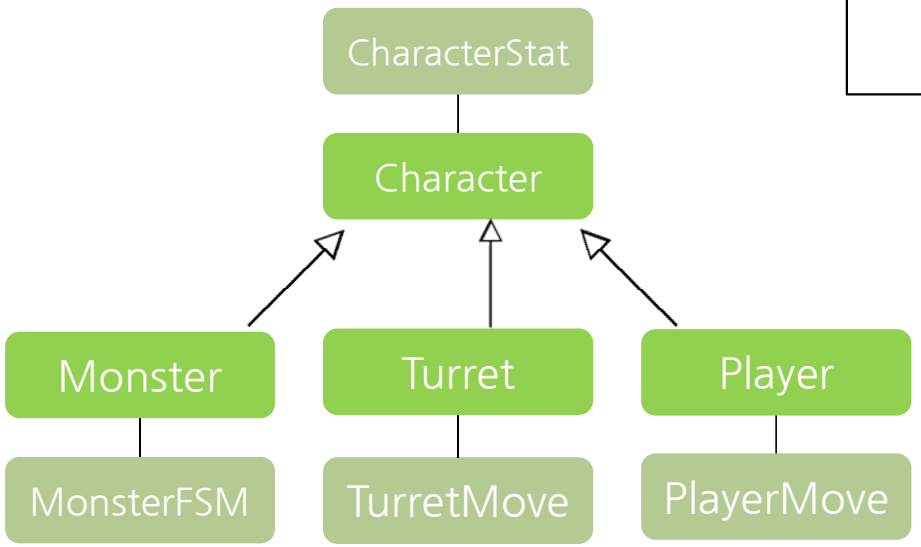
2. 설계 - 클래스 구조도 - 전투

Manager, Pool
매니저는 객체들을 관리합니다.
Pool은 오브젝트의 풀링을 담당합니다.
기본적으로 뒤에 Pool이 붙은 클래스는
ObjectPool 클래스를 상속받습니다.
또한, 몇몇 매니저 클래스와, 풀들은 싱글톤 패턴을 통해
객체를 한 개만 생성하여 혼란을 방지합니다.



DataManager는 게임의 저장 정보를
관리하는 기능을 가진 싱글톤 클래스 입니다.
돈, 각 스텟의 레벨, 터렛 배치를
PlayerPrefs를 통해 세이브, 로드 할 수 있습니다.
각 객체들은 이 클래스를 통해 저장 정보들을
얻어올 수 있습니다.

2. 설계 - 클래스 구조도 - 전투



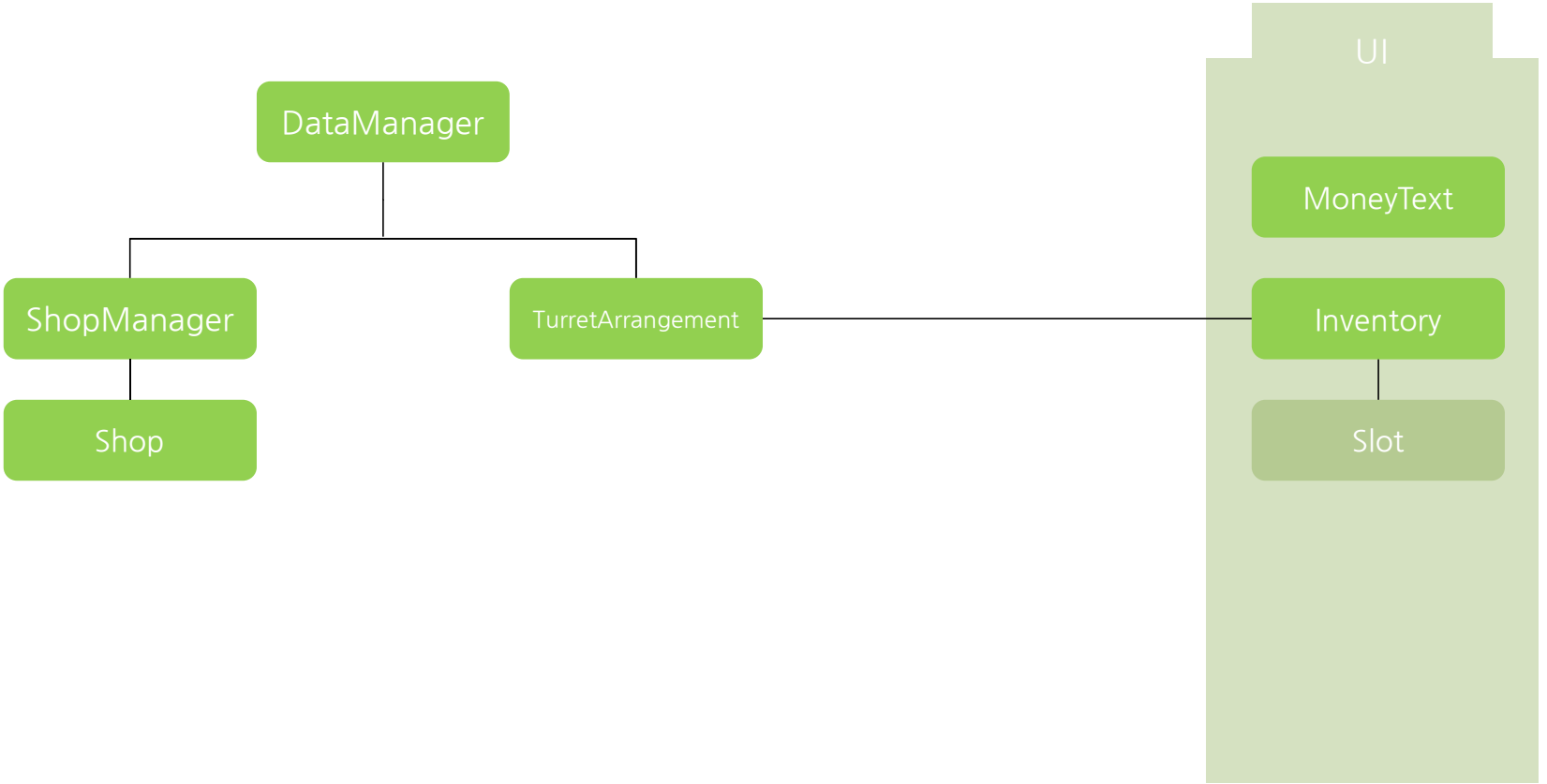
Character

캐릭터는 기본 내용(HP, 공격력 등)을 가졌고,
몬스터, 터렛, 플레이어 클래스가 상속받습니다.
각자가 움직임을 담당하는 클래스로
클래스 이름 뒤에 Move가 붙은 클래스를 가집니다.

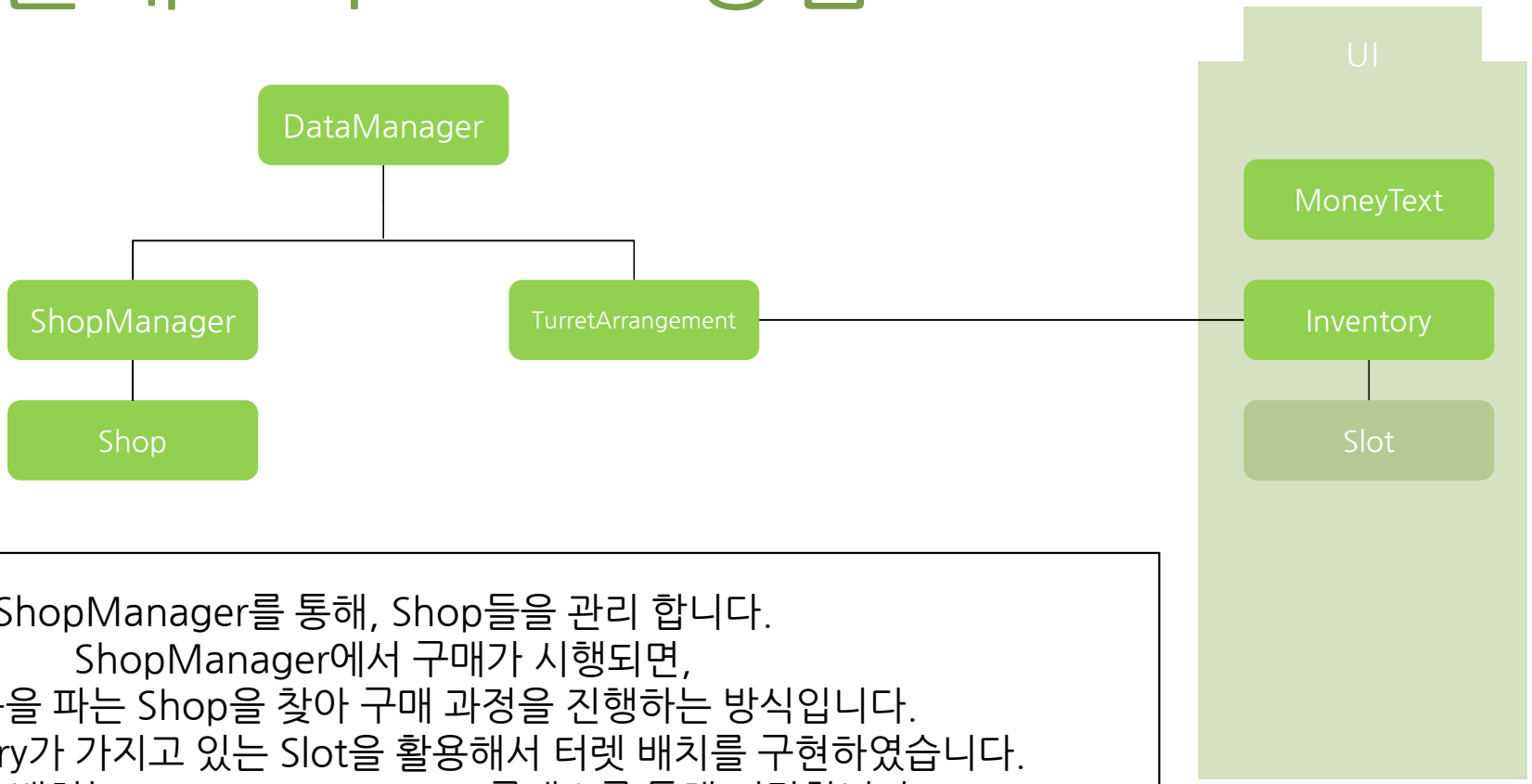
Monster는
FSM을 통해 자동으로
상태를 바꾸면서
행동합니다.

2. 설계 - 클래스 구조도 - 상점

2. 설계 - 클래스 구조도 - 상점



2. 설계 - 클래스 구조도 - 상점



ShopManager를 통해, Shop들을 관리 합니다.
ShopManager에서 구매가 시행되면,
맞는 품목을 파는 Shop을 찾아 구매 과정을 진행하는 방식입니다.
그리고 Inventory가 가지고 있는 Slot을 활용해서 터렛 배치를 구현하였습니다.
그리고 그 배치는 TurretArrangement 클래스를 통해 저장합니다.
모든 정보는 DataManager를 통해 저장합니다.

3. 주요 구현 부 - FSM

- FSM (Finite State Machines)
  유한 상태 기계를 활용하여
  Idle, 정찰, 공격, 사망의 상태를 갖게 구현.

```
public class FiniteState
{
    #region Values
    public EState stateId;  // 현재 상태 아이디
    public Dictionary<EEvent, EState> transList = new Dictionary<EEvent, EState>();
    // 어느 이벤트 발생시 어떤 상태로 바꾸어야 하는지를 Dictionary로 갖고 있는다.
    #endregion

    public FiniteState(EState _stateId)
    {
        stateId = _stateId;
    }

    // 이벤트 하나에 상태 하나 연결.
    public void AddTransition(EEvent _inputEvent, EState _outputState)
    {
        transList[_inputEvent] = _outputState;
    }

    // 이벤트에 따른 출력 상태를 리턴
    public EState OutputState(EEvent _inputEvent)
    {
        if (!transList.ContainsKey(_inputEvent))
        {
            string str = string.Format("Event = {0} state = {1}", _inputEvent, stateId);
            Debug.LogError("Input Event is not vaild : " + str);
        }
        return transList[_inputEvent];
    }
}
```

```
public class MonsterFSM
{
    #region Components
    private FiniteState currentState;   // 현재 상태
    private Dictionary<EState, FiniteState> stateList = new Dictionary<EState, FiniteState>();
    // 상태키에 따른 상태를 갖고 있는 Dictionary
    #endregion

    // 시작 상태 - (이벤트) -> 출력 상태 로 연결
    public void AddStateTransition(EState _stateId, EEvent _inputEvent, EState _outputStateId)
    {
        FiniteState state = null;

        if (stateList.ContainsKey(_stateId))    // 시작 상태가 사전에 이미 있으면 재활용
        {
            state = stateList[_stateId];
            state.AddTransition(_inputEvent, _outputStateId);
        }
        else
        {
            state = new FiniteState(_stateId);
            state.AddTransition(_inputEvent, _outputStateId);
            stateList.Add(_stateId, state);
        }
    }
    // 현재상태 강제 설정
    public void SetCurrentState(EState _stateId)
    {
        currentState = stateList[_stateId];
    }
    // 상태 전환.
    public void StateTransition(EEvent _eventId)
    {
        EState outputStateId = currentState.OutputState(_eventId);
        currentState = stateList[outputStateId];
    }
```

3. 주요 구현 부 - FSM

```
// 모든 상태들은 MonsterState를 상속 받는다.
public abstract class MonsterState
{
    #region Values
    public int aniIndex;    // 상태마다 각자의 애니메이션 번호
    public EState state;    // 상태 -> 각각의 상태 객체들은 상태 키와 연동됨.
    #endregion
    #region Components
    protected Monster parentObj;    // 부모 객체 -> 상태를 가지고 있는 객체 (몬스터)
    #endregion

    public MonsterState() { }
    public MonsterState(Monster _parent, int _aniIndex)
    {
        parentObj = _parent;
        aniIndex = _aniIndex;
    }
    // 상태에 돌입 했을 때
    public abstract void Enter(Character _target);
    // 상태에서 해야할 행동 (Update에서 실행)
    public abstract void Handle(Character _target);
}
```

상태 중 하나인 정찰상태 예.

```
public class PatrolState : MonsterState
{
    #region Values
    Vector3 destPos = Vector3.zero; // 목적지 좌표
    #endregion

    public PatrolState(Monster _monster, int _aniIndex) : base(_monster, _aniIndex)
    {
        parentObj = _monster;
        state = EState.STATE_PATROL;
        MakeNewDestPos();
    }

    public override void Enter(Character _target)
    {
        MonsterManager.Instance.AttackingMonsterList.Remove(parentObj);
        parentObj.MyNavMeshAgent.isStopped = false;
        MakeNewDestPos();
        Patrol();
    }
    public override void Handle(Character _target)
    {
        CheckTargetIsInRange();
        CheckArrive();
    }
```

```
public class Monster : Character
{
    public void Update()
    {
        currentState.Handle(theTarget);
        RenderSetting();
    }
```

MonsterFSM을 가지고 있는 객체는, Handle()을 업데이트에서 매 프레임 마다 실행시킵니다.

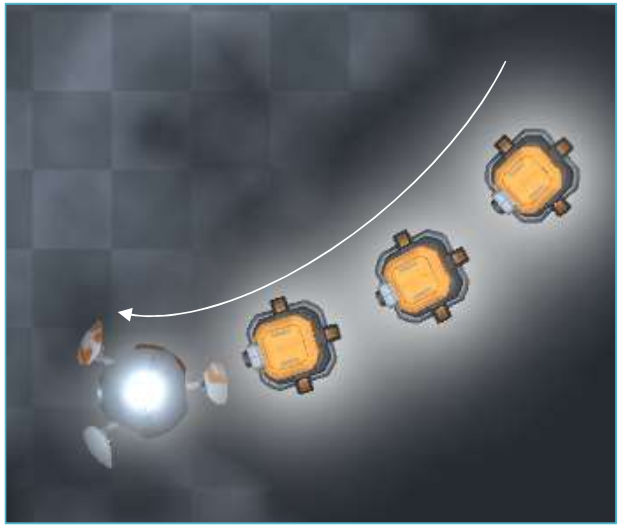
3. 주요 구현 부 - FSM

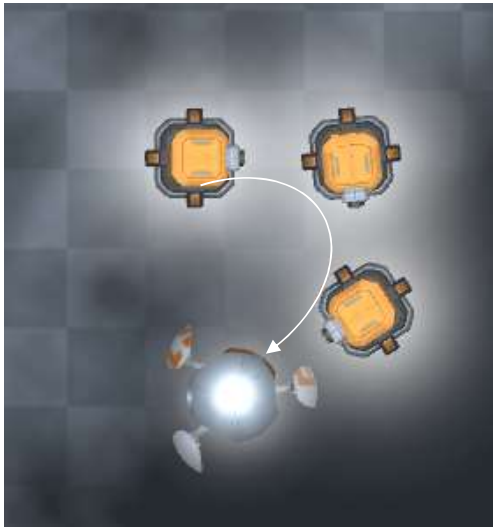
```
for (int i = 0; i< nMobCount; i++)
{
    MonsterFSM temp = new MonsterFSM();

    temp.AddStateTransition(EState.STATE_IDLE, EEvent.EVENT_PATROL, EState.STATE_PATROL);
    temp.AddStateTransition(EState.STATE_IDLE, EEvent.EVENT_FINDTARGET, EState.STATE_ATTACK);
    temp.AddStateTransition(EState.STATE_IDLE, EEvent.EVENT_DEAD, EState.STATE_DEAD);

    temp.AddStateTransition(EState.STATE_PATROL, EEvent.EVENT_FINDTARGET, EState.STATE_ATTACK);
    temp.AddStateTransition(EState.STATE_PATROL, EEvent.EVENT_STOPWALK, EState.STATE_IDLE);
    temp.AddStateTransition(EState.STATE_PATROL, EEvent.EVENT_DEAD, EState.STATE_DEAD);

    temp.AddStateTransition(EState.STATE_ATTACK, EEvent.EVENT_LOSTTARGET, EState.STATE_IDLE);
    temp.AddStateTransition(EState.STATE_ATTACK, EEvent.EVENT_STOPWALK, EState.STATE_IDLE);
    temp.AddStateTransition(EState.STATE_ATTACK, EEvent.EVENT_DEAD, EState.STATE_DEAD);

    temp.AddStateTransition(EState.STATE_DEAD, EEvent.EVENT_REVIVE, EState.STATE_IDLE);

    temp.SetCurrentState(EState.STATE_PATROL);
    // 몬스터 FSM을 초기화 하는 작업
```

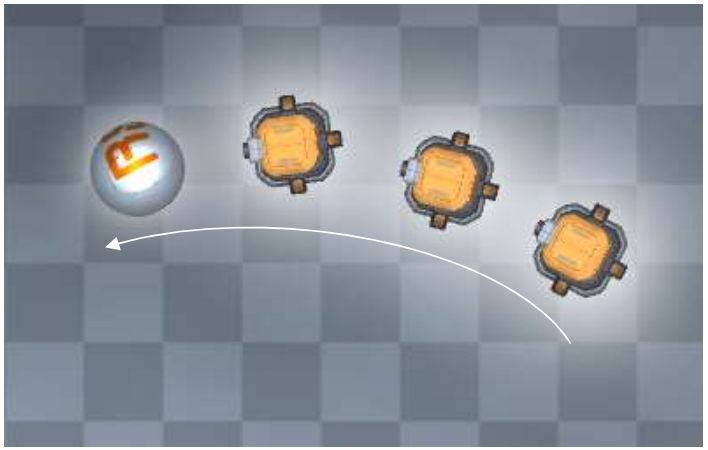
적을 생성할 때 다음과 같이 FSM을 초기화 해줍니다.

3. 주요 구현 부 - TurretMove

뱀 게임의 꼬리처럼 앞 대상을 쫓아다니는 터렛의 움직임을 구현







3. 주요 구현 부 - TurretMove

쫓아가는 대상을 계속해서 쫓아가지만, 거리는 유지하는 방식으로 구현하였습니다.

```
private IEnumerator StartChaseTarget()
{
    yield return new WaitUntil(() => CalculateDistance() >= distance);  // 쫓아가는 대상과 일정 거리가 벌어지면 코루틴 실행

    dir = chasingTarget.transform.position - transform.position;
    transform.Translate(dir.normalized * Time.deltaTime * speed);   // 앞 대상을 향하는 벡터 대로 이동

    StartCoroutine(StartChaseTarget()); // 코루틴 반복
}
private float CalculateDistance()
{
    return Vector3.Distance(transform.position, chasingTarget.transform.position);  // 쫓아가는 대상과의 거리 반환
}
public void LookChasingTarget() // 쫓아가는 대상을 바라보게 함.
{
    Vector3 temp = chasingTarget.transform.position;
    temp.y = turretModel.transform.position.y;
    turretModel.plate.transform.LookAt(temp);
    turretModel.head.transform.LookAt(temp);
}
```

4. 최적화 - Object Pooling

게임 오브젝트의 생성-삭제 과정은 작은 일이지만,
대량으로 진행되면 CPU에 과부하가 갈 것이고,
프레임 드롭을 불러올 것이기 때문에.

오브젝트 풀을 만들어 싱글톤 패턴으로 관리합니다.

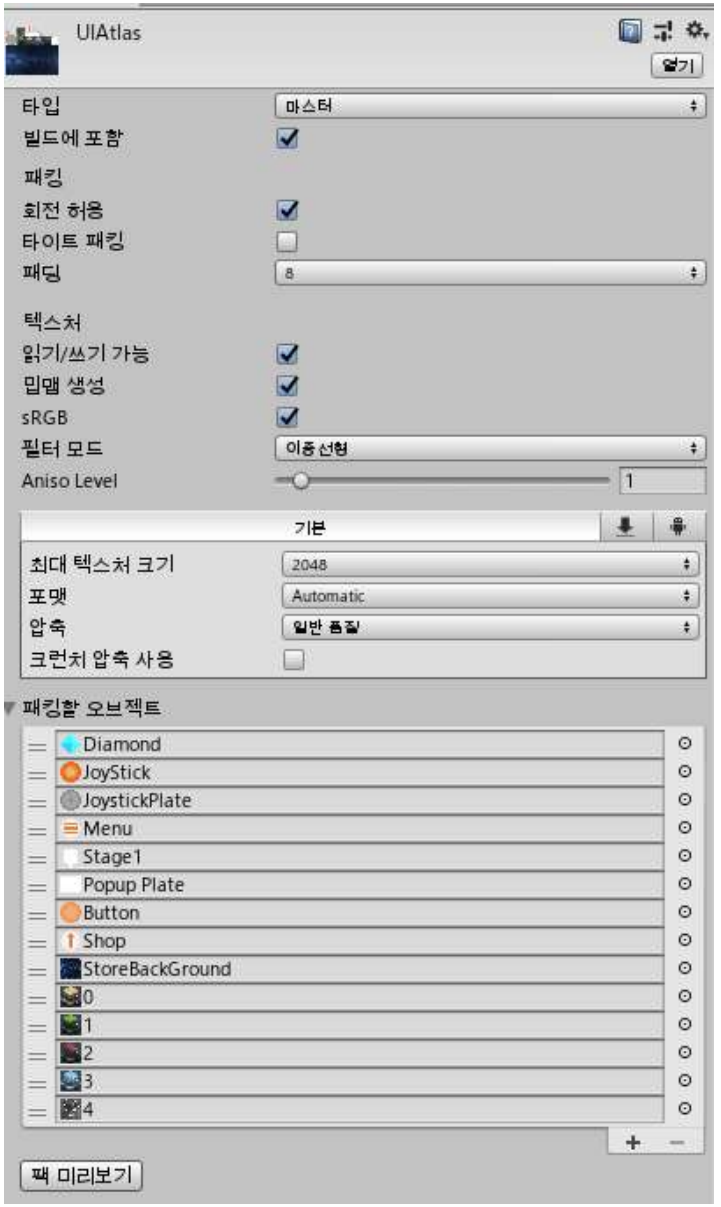
모든 오브젝트 풀들은 ObjectPool 클래스를 상속 받아
간편하게 오브젝트 풀을 생성할 수 있도록 하였습니다.

```
// 풀 생성
private void PrepareObject(int _prepareCount)
{
    StringBuilder strBulder = new StringBuilder();

    T temp;
    for (int i = 0; i < _prepareCount; i++)
    {
        // Create Bullet
        temp = null;
        temp = Instantiate(originObject, objParent.transform);

        // Naming Bullet
        strBulder.Clear();
        strBulder.Append(objName);
        strBulder.Append((objectList.Count / 10) * 10 + i);
        temp.name = strBulder.ToString();

        // De-Activate Bullet
        temp.gameObject.SetActive(false);

        // Add to List
        objectList.Add(temp);
    }
}
// 풀에서 오브젝트 떼어 주기.
public T GetObject(bool active = true)
{

    T newActiveObject = objectList[activeCount];

    newActiveObject.gameObject.SetActive(true);

    activeCount++;
    if (activeCount >= objectList.Count)
        activeCount = 0;

    return newActiveObject;
}
```

4. 최적화 - Sprite Atlas

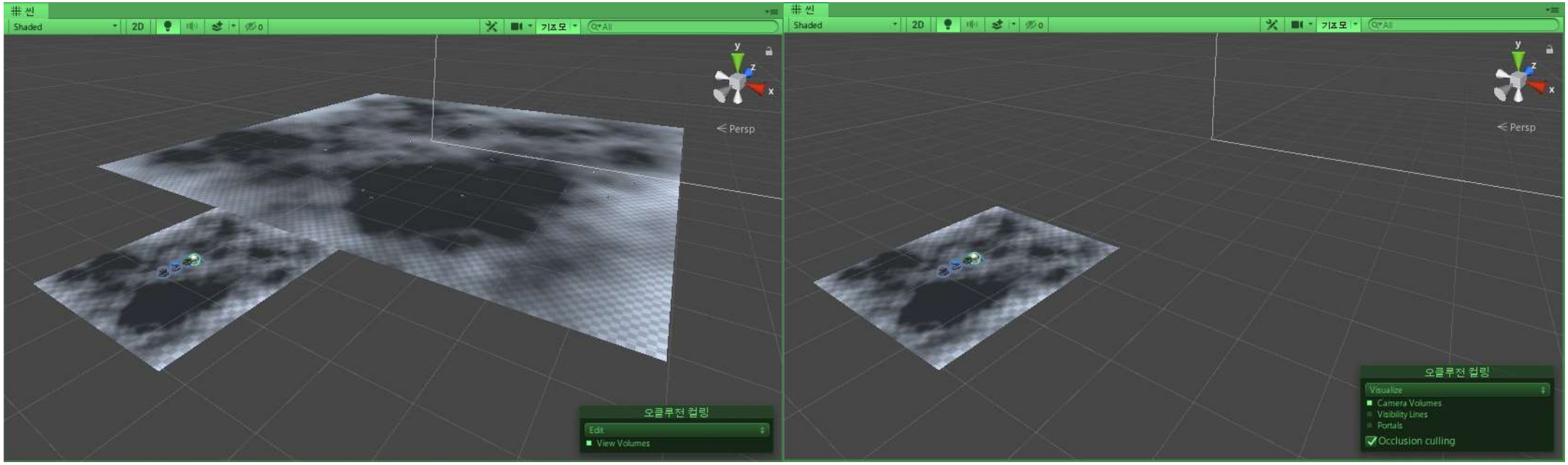
게임에서 사용되는 수많은 Sprite들을 개별적으로 사용하는 것은 성능을 많이 잡아먹기 때문에,

이 Sprite들을 하나로 모아 Sprite Atlas를 만들어서 드로우 콜을 줄여서 최적화를 했습니다.



4. 최적화 - 오클루전 컬링

오클루전 컬링을 활용하여, 시야 밖의 오브젝트들은 렌더링 하지 않도록 설정하여,
필요 없는 렌더링 작업을 줄여 최적화 하였습니다.



4. 최적화 - 코루틴 활용

```
void ChaseTarget()
{
    float distance = Vector3.Distance(transform.position, chasingTarget.transform.position);
    dir = chasingTarget.transform.position - transform.position;

    if (distance ≥ 1f)
    {
        transform.Translate(dir * Time.deltaTime * speed);
    }
}
```

원래 터렛들이 앞의 대상을 쫓아가는 코드는 위와 같이 구현했었습니다.
위의 함수를 업데이트 함수에서 돌려서, 매 프레임 거리를 계산하고, 거리가 멀면 이동을 진행하는 방식이었습니다.

```
private IEnumerator StartChaseTarget()
{
    yield return new WaitUntil(() ⇒ CalculateDistance() ≥ distance);  // 쫓아가는 대상과 일정 거리가 벌어지면 코루틴 실행

    dir = chasingTarget.transform.position - transform.position;
    transform.Translate(dir.normalized * Time.deltaTime * speed);   // 앞 대상을 향하는 벡터 대로 이동

    StartCoroutine(StartChaseTarget()); // 코루틴 반복
}
```

그래서 Update에서 돌리지 않고, 위와 같이 코루틴으로 구현하였습니다.
이렇게 해서, 거리가 벌어질 때 까지 코루틴은 휴식하도록 하였습니다.

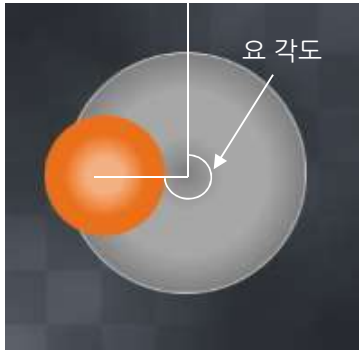
4. 최적화 - 리소스 로드

```
private void Awake()    // 최적화를 위해 리소스들을 미리 로드 해놓음
{
    DontDestroyOnLoad(Instance);
    if (Instance ≠ this)
        Destroy(gameObject);

    StringBuilder tempStr = new StringBuilder();

    hpBar = Resources.Load("UI/HPBar", typeof(HPBar)) as HPBar;
    bullet = Resources.Load("Bullet") as GameObject;
    monster = Resources.Load("Monster") as GameObject;
    for (int i = 0; i < turretSpriteCount; i++)
    {
        tempStr.Append("TurretSprite/");
        tempStr.Append(i.ToString());
        spriteList.Add(Resources.Load(tempStr.ToString(), typeof(Sprite)) as Sprite);
        tempStr.Clear();
    }
    for(int i = 0; i < turretModelCount; i++)
    {
        tempStr.Append("TurretModel/");
        tempStr.Append(i.ToString());
        turretModelList.Add(Resources.Load(tempStr.ToString(), typeof(TurretModel)) as TurretModel);
        tempStr.Clear();
    }
}
```

ResourceManager를 싱글톤으로 만들고, DontDestroyOnLoad로 한 번만 로딩하게 해주었습니다.

리소스를 불러오는 건 무거운 작업이기에, 매번 리소스를 로딩하지 않고,
이 클래스를 통해 미리 불러와진 리소스를 가볍게 가져올 수 있도록 구현하였습니다.

5. 문제 해결 과정 - 조이스틱 각도 계산



위 방향을 기준으로, 조이스틱의 각도를 구하려고 했습니다.

이 각도를 구해서, 플레이어의 로테이션 값을 조절해,
자연스러운 움직임을 구현하고 싶었습니다.

유니티에 함수가 있을 거라 생각했지만, 없어서 헤맸습니다.

5. 문제 해결 과정 - 조이스틱 각도 계산

```
private void RotationAngleInDegrees()
{
    float theta = Mathf.Atan2(stick.transform.position.y - plate.transform.position.y, stick.transform.position.x - plate.transform.position.x);

    theta -= Mathf.PI / 2.0f;

    float _angle = (theta * Mathf.Rad2Deg);

    if (_angle < 0) _angle += 360;

    _angle -= 360;

    angle = Mathf.Abs(_angle);  // 조이스틱의 중심을 축으로 한뒤, 위 방향을 기준으로 내가 터치한 부분 각도를 계산하는 과정.
}
```

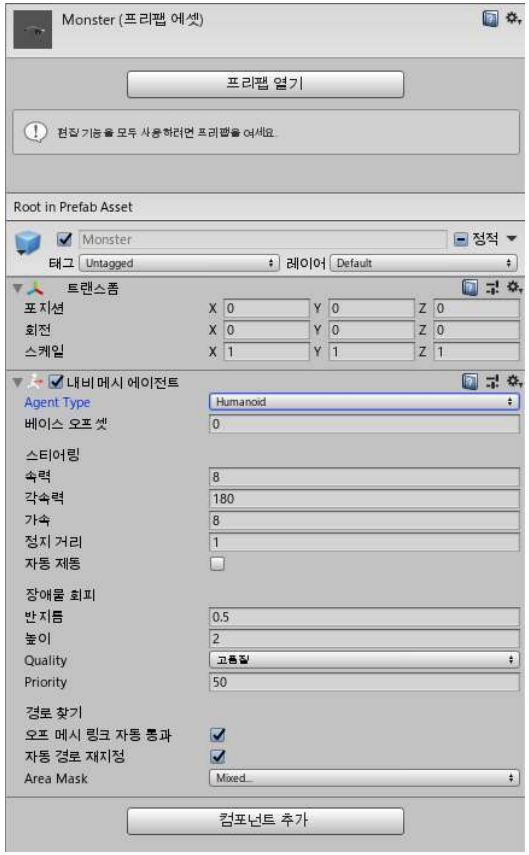
고등학교 미적분 시간에 배운 삼각함수를 활용하면 될 것 같아서
삼각 함수를 활용해 라디안 값을 구해서
조이스틱의 각도를 구할 수 있었습니다.

5. 문제 해결 과정 - 적의 이동방식

```
parentObj.transform.position = Vector3.MoveTowards(parentObj.transform.position, destPos, Time.deltaTime * parentObj.Speed);
```

처음엔 적이 이동하는 방식을 위와 같이 MoveTowards로 이동하게 했습니다.
이러다 보니 몬스터들이 지나가다가 부딪혀도 뚫고 지나가버리고,
캐릭터를 쫓아갈 때엔, 뮤탈리스크처럼 뭉쳐버리는 일이 생겼습니다.

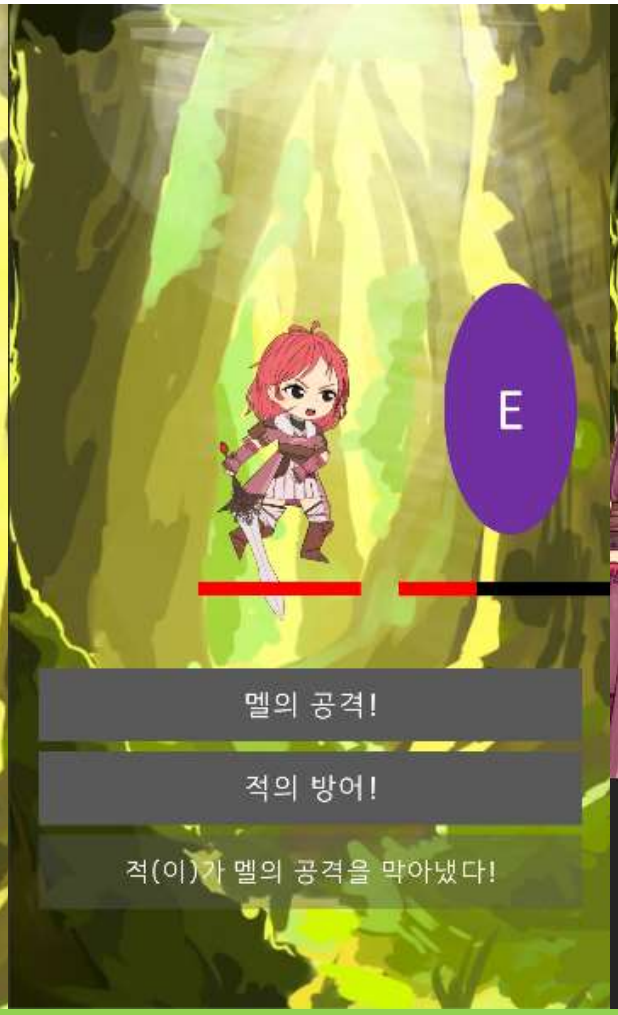
5. 문제 해결 과정 - 적의 이동방식



Rigidbody와 Collider를 적용하면, 게임이 무거워 지고,
그렇다고 좌표로 계산을 하자니 감이 안 잡혔습니다.

방법을 강구하다가, Navigation기능이 떠올랐습니다.

적용을 하자, 정찰을 하면서도 알아서 비껴가고,
플레이어를 쫓아갈 때에도 뭉치지 않게 되었습니다.



TEIA

안드로이드의 턴제 RPG입니다. 스토리 텔링에 초점을 둔 게임입니다.

0. 게임 설명

제목 - TEIA (테이아)

장르 - 턴제 RPG

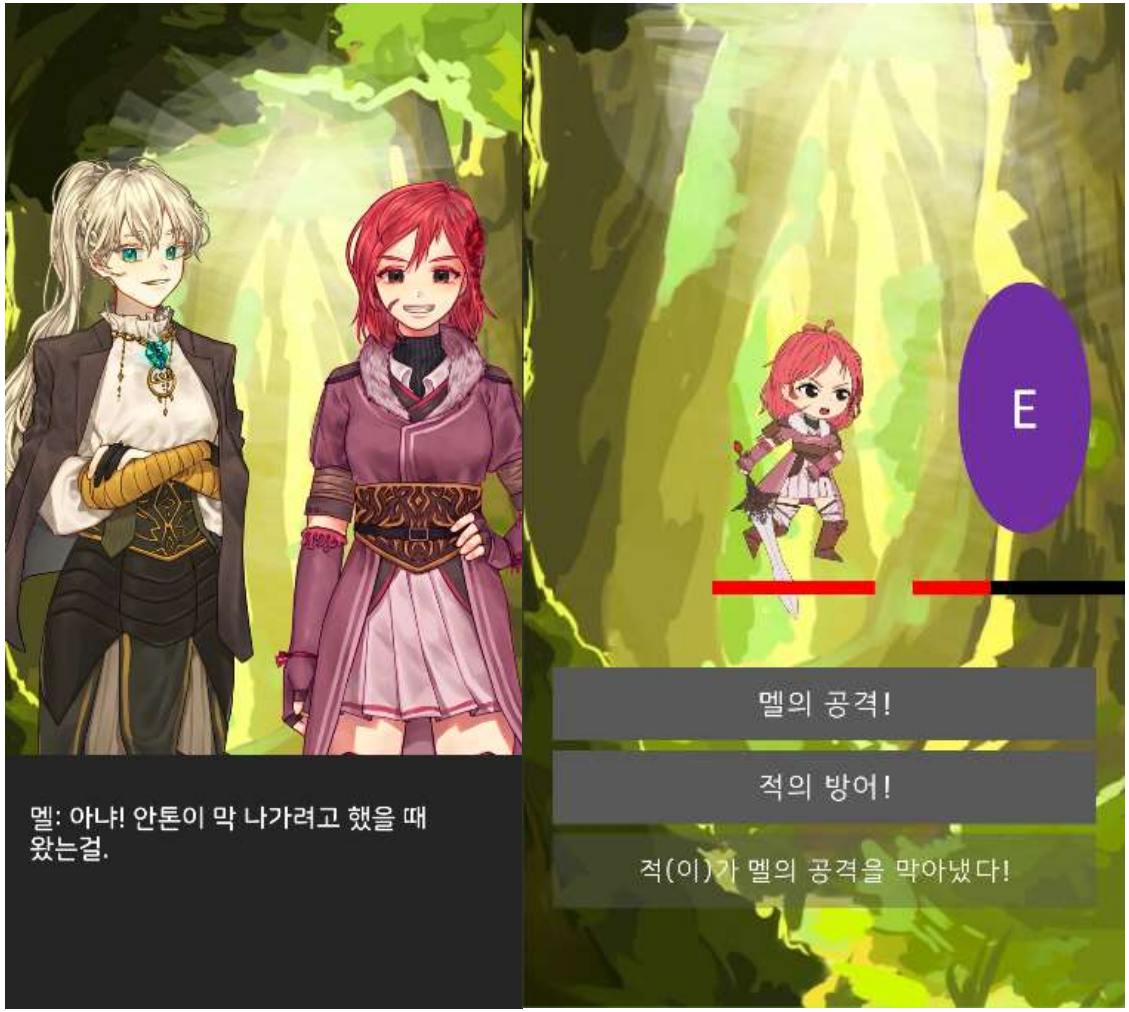
플랫폼 - 안드로이드

플레이 영상 - https://blog.naver.com/tlaryska12/221694602255

0. 게임 설명

테이아는 스토리가 메인이 되는 게임입니다.
메인 퀘스트를 진행하면서 전투를 벌이고,
스토리를 진행시킬 수 있습니다.

또한, 전투를 통해 얻은 재화로 캐릭터의 레벨을
올리고, 강화를 할 수도 있습니다.



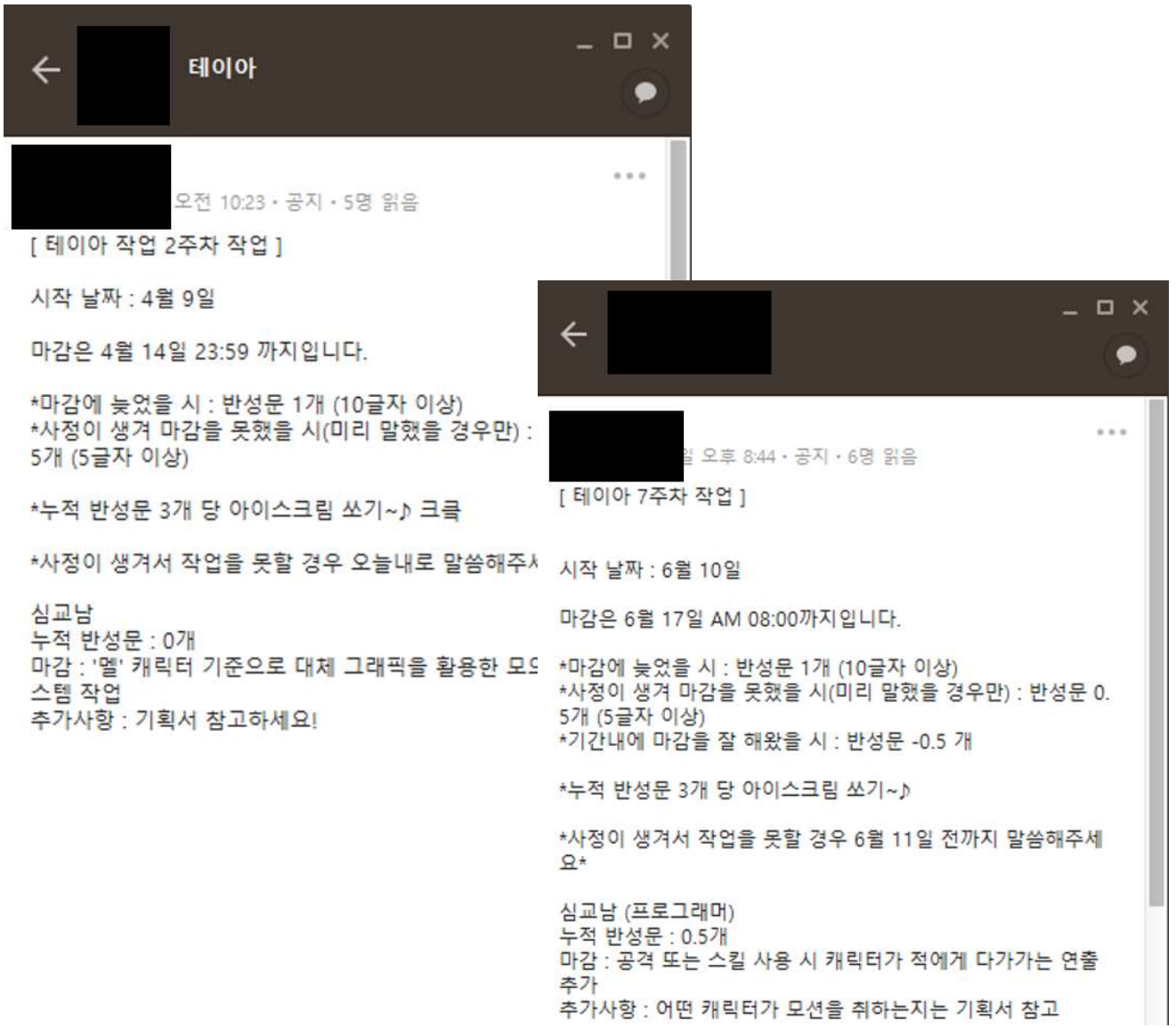
1. 개발 일정

재학 기간 동안, 자투리 시간에 조금씩 개발하였습니다.
팀원들과 팀 프로젝트로 진행하였습니다.

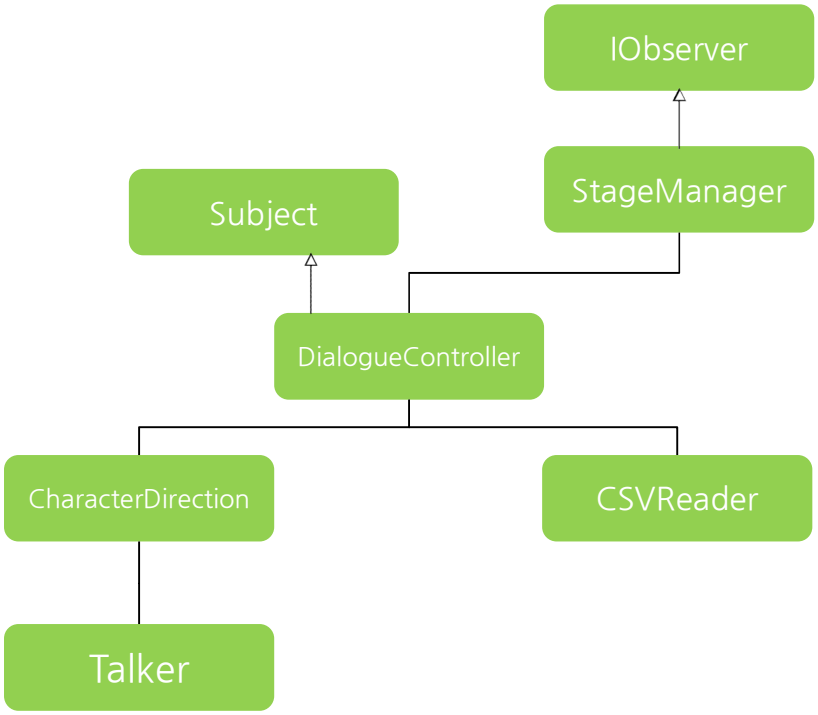
일정 진행 방식은 기획자 님이 한 주치 일거리를 주시면,
그 일정대로 활동하는 방식이었습니다.

당시, 팀원이었던 친구들이 대학과 취업 사이에서
진로를 고민하던 시기였기에, 학업에 지장이 가지 않게
일주일도 많을 정도로 적은 양씩 일을 받았습니다.

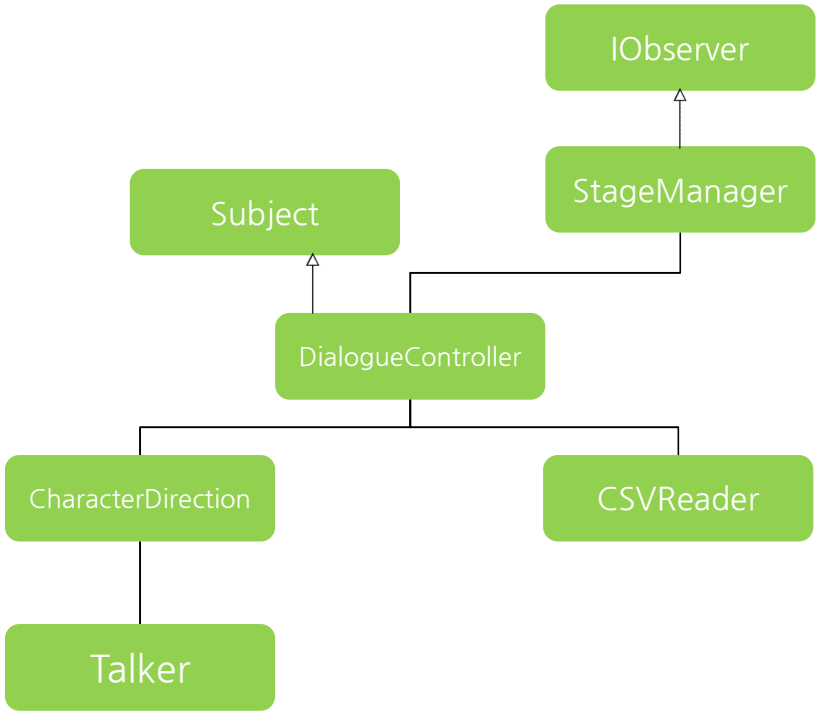
그래서 꽤나 긴 기간 동안 프로젝트가 진행되었습니다.



2. 설계 - 스토리 연출


IObserver
StageManager
Subject
DialogueController
CharacterDirection
CSVReader
Talker

2. 설계 - 스토리 연출

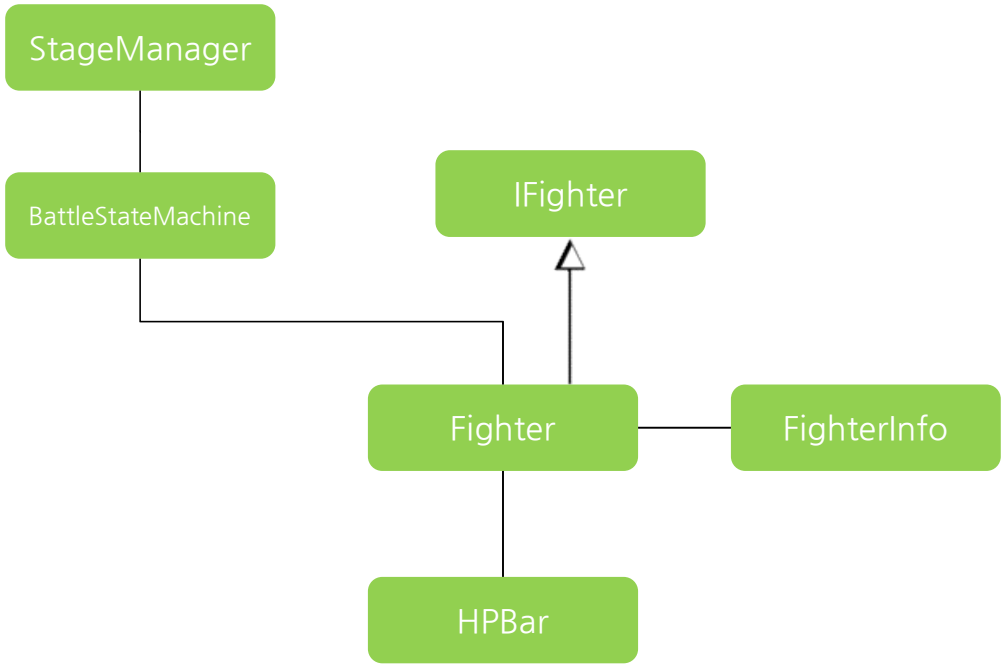


StageManager를 통해, 스토리가 나오는 화면과,
전투를 하는 화면을 조절합니다.

DialogueController를 통해 스토리 보드를 재생합니다.
전투 씬으로 넘어가야 하거나 스토리가 끝나면,
옵저버 패턴을 통해 StageManager에 공지합니다.

CSVReader를 통해 csv파일로 저장된 스토리 보드를
게임 시작 시에 파싱한 뒤, 로딩해서 리스트에 저장합니다.

2. 설계 - 전투

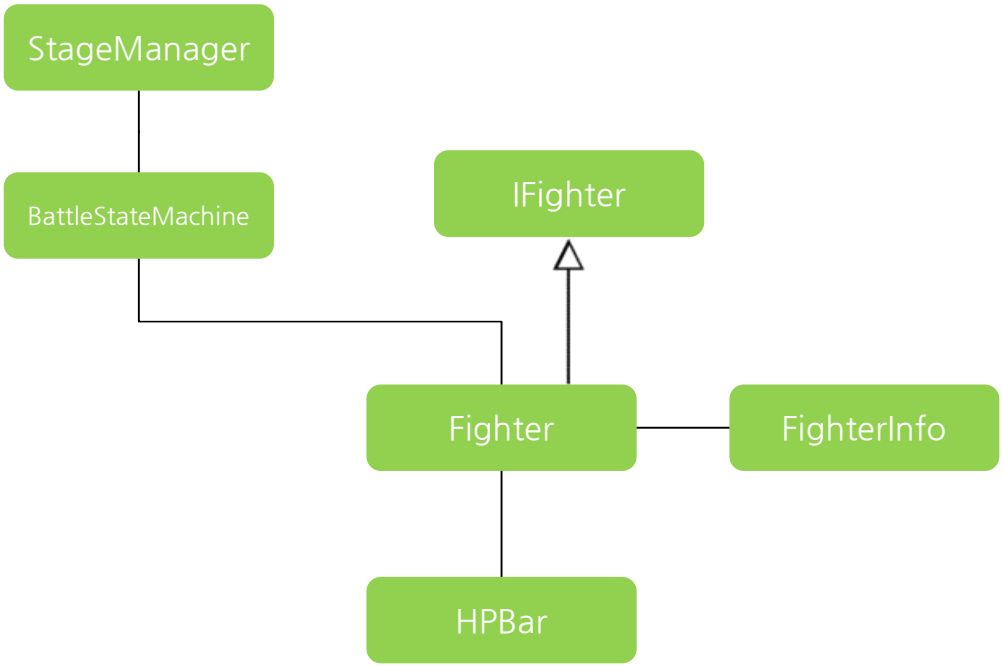

StageManager
BattleStateMachine
IFighter
Fighter
FighterInfo
HPBar

2. 설계 - 전투

StageManage를 통해, 전투 화면으로 넘어가야 할 때엔
BattleStateMachine을 활용해서 전투를 진행시킵니다.

BattleStateMachine은 FSM으로 구성되어 있습니다.

Fighter는 플레이어와 몬스터의 기능을 담당합니다.
FighterInfo를 통해 Fighter별로 각각의 스탯을 부여합니다.



3. 주요 구현 부 - 스토리 보드 재생

RPG게임에 있는 텍스트가 나오는 간단한 대화 창과 캐릭터가 나오고, 사라지는 연출을 구현하였습니다.

3. 주요 구현 부 - 스토리 보드 재생

Number	BackGround	Talker	Face	Speech	Direction	A	B	C
0	프롤로그			흐레스트 건국령 258년.				
1				19년 전 부터 이어져온 흐레스트 왕국과 마왕군의 기나긴 전쟁은 여전히 끝날 기미가 보이지 않았다.				
2				왕국을 향한 마왕의 이유모를 분노는 사람들을 불안에 떨게 하였으며 국가의 정세는 계속해서 흔들렸다.				
3				이에 왕국에서는 마왕군에 대적할 기사단을 전국적으로 모집하기 시작했다.				
4				그 중 '쉬폴리 기사단'은 국왕 흐레스트 4세를 호위하는 직속 친위대로, 모든 수련생들이 동경하는 기사단이었다.				
5				한편 수도에서 가장 멀리 떨어져 비교적 평화로운 왕국의 변두리. 엘리파 마을.				
6				특산물인 사과 외에는 별다른 특징이 없는 이곳에도 쉬폴리 기사단의 입단을 위해 땀 흘리며 수련하는 세명의 수련생들이 있었다...				
7	멜의 집	안톤	무심함	멜.	1			
8		멜	웃음	앗, 안톤! 무슨 일이야?	05월 01일	안톤	멜	
9		안톤	무심함	수련시간 거의 다 됐는데 잊어버리고 있는 것 같길래 데리러 왔어.				
10		멜	의문	아냐! 이제 막 가려고 했는걸!				
11		안톤	무심함	아닌 것 같은데. 아직 신발도 안 신고 있잖아?				
12		멜	놀람	헉.				
13		안톤	웃음	누굴 속이려고. 빨리 가자.	08월 01일	멜	안톤	
14		멜	당황	으으. 정말, 좀처럼 속아주는 걸 못 본다니까!	11			
15	엘리파 마을	멜	웃음	(나는 기사 연습생 멜.)				
16		멜		(부모님이 계신 수도의 쉬폴리 기사단에 입단하기 위해 친구들과 함께 수련중이다.)				
17		안톤	웃음	루나. 멜 데려왔어.	1	안톤		
18		루나	웃음	수고했어. 안녕 멜~ 오늘도 늦잠 잤지?	1	루나		
19		멜	진지함	아냐! 안톤이 막 나가려고 했을 때 왔는걸.	06월 01일	루나	멜	
20		루나	무표정	정말?				
21		안톤	무표정	아마. 그래도 현관에는 나와 있었어.	10	루나	멜	안톤
22		멜	웃음	(안톤은 태어나서부터 거의 옆에서 함께 자란 나의 단짝친구다.)				
23		멜		(마법에 재능이 있어서, 기사단의 마법사를 목표로 수련하고 있다.)				
24		멜		(표정변화도 별로 없고 무뚝뚝해 보이지만 속은 꼭 그렇지만은 않다는 걸 나는 잘 알고 있어!)				
25		루나	무표정	그럼 굳이 안톤을 안 보내도 괜찮았겠네.	9	멜	안톤	루나
26		멜	웃음	(루나도 굉장히 오래 본 친구다.)				
27		멜		(어렸을 때 숲에 혼자 있던 것을 나와 안톤이 발견해 마을로 데리고 왔다.)				
28		멜		(그 후로 루나도 함께 기사단의 궁수를 목표로 열심히 노력하고 있다.)				
29		멜	밝게 웃음	(그리고 사과파이를 굉장히 잘 만든다!)				
30		멜	기대	루나! 오늘도 수련 끝나고 사과 파이 만들어 줄거지?				
31		루나	무표정	집에 사과가 남아있는지 모르겠는데?				
32		멜	기대	없으면! 나랑 안톤이 사올게!				
33		안톤	당황	나도??	10	루나	멜	안톤
34		멜	웃음	당연하지. 안톤도 먹을거잖아.				
35		안톤	무심함	.....				
36		안톤	웃음	그래.				
37		루나	무표정	좋아. 그럼 그렇게 하기로 하고, 슬슬 시작할 시간이야.	9	멜	안톤	루나
38		멜	웃음	오늘도 힘내자!				
39		전투						

기획자 분한테 스토리보드를 CSV파일로 받고, 이 파일을 파싱해서 딕셔너리로 저장했습니다. 각 스테이지별 스토리보드는 리스트에 담아 관리하였습니다.

3. 주요 구현 부 - 스토리 보드 재생

```
private void NextDialogue()
{
    if ((int)storyBoard[dialogueCount]["Number"] == -1)
    {
        Debug.Log("dialogue ends");
        return;
    }
    if (storyBoard[dialogueCount]["Talker"].ToString().Equals("전투"))
    {
        Notify(EVENT.GOBATTLE);
        dialogueCount++;
        return;
    }
    if(storyBoard[dialogueCount]["Direction"].ToString().Length != 0)
    {
        theCharacterDirection.PlayDirection(
            storyBoard[dialogueCount]["Direction"].ToString(),
            storyBoard[dialogueCount]["A"].ToString(),
            storyBoard[dialogueCount]["B"].ToString(),
            storyBoard[dialogueCount]["C"].ToString()
            );
    }
```

```
    StringBuilder _text = new StringBuilder();

    if (storyBoard[dialogueCount]["Talker"].ToString().Length != 0)
    {
        currentTalker = storyBoard[dialogueCount]["Talker"].ToString();

        //if (storyBoard[dialogueCount]["Face"].ToString().Length != 0)
        //{
        //    theCharacterDirection.ChangeFace(
        //        currentTalker,
        //        storyBoard[dialogueCount]["Face"].ToString()
        //        );
        //}
        _text.Append(currentTalker);
        _text.Append(": ");
    }
    _text.Append(storyBoard[dialogueCount]["Speech"].ToString());

    StartCoroutine(ShowText(_text.ToString()));

    dialogueCount++;
}
```

이후 받아온 스토리 보드를, DialogueController클래스에서 위 함수의 내용과 같이 재생해줍니다.
(주석 친 부분은, 캐릭터의 표정 스프라이트가 디자이너 분으로 부터 나오지 않아서 미리 구현해 놓은 부분입니다.)

3. 주요 구현 부 - 전투

처음 시작하면, 플레이어가 할 행동을 선택하고,
조금 뒤, 적도 행동을 자동으로 선택합니다.

이후 그 플레이어와 적의 선택을 비교하여,
결과를 도출해냅니다.

이 단순한 프로세스를 무리 없이 구현해야 했습니다.

3. 주요 구현 부 - 전투

CleanerBall에서 구현했던 FSM과 다르게,
매우 정석적인 FSM을 구현하였습니다.

대기, 플레이어 턴, 적 턴, 결과 계산, 전투 종료(보상&결과 계산)
총 다섯 가지의 상태를 가지고 있습니다.

```
public abstract class BattleState
{
    public abstract void Enter(BattleStateMachine bsm);
    public abstract void Handle(BattleStateMachine bsm);
    public abstract void Exit(BattleStateMachine bsm);
}

public class WaitingState : BattleState
{
    public override void Enter(BattleStateMachine bsm) ...
    public override void Handle(BattleStateMachine bsm) ...
    public override void Exit(BattleStateMachine bsm) ...
}
public class PlayerTurnState : BattleState
{
    private Fighter targetFighter;
    private float timeCount;

    public override void Enter(BattleSta
    public override void Handle(BattleSt
    public override void Exit(BattleStat
}
public class EnemyTurnState ...
public class CalculatingState ...
public class ResultingState ...
```

```
public void Update()
{
    currentState.Handle(this);
}
public void ChangeState(BattleState _state)
{
    currentState.Exit(this);

    currentState = _state;

    currentState.Enter(this);
    currentState.Handle(this);
}
```

3. 주요 구현 부 - 전투와 스토리 연출 조종

```
public class StageManager : MonoBehaviour, IObserver
{
    void IObserver.OnNotify(EVENT _event)
    {
        switch (_event)
        {
            case EVENT.GOBATTLE:
                PlayBattle();
                break;
            case EVENT.ENDBATTLE:
                PlayDialogue();
                break;
        }
    }
```

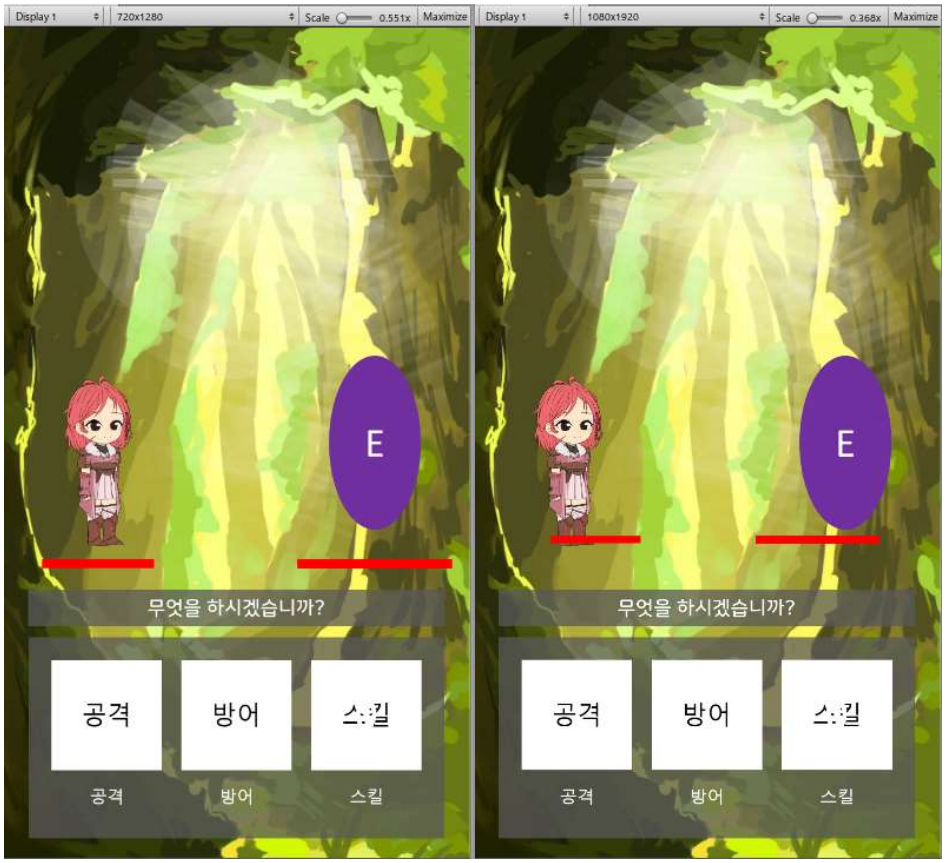
```
public class ResultingState : BattleState
{
    public override void Enter(BattleStateMachine bsm)
    {
        bsm.ShowEndPanel();
        bsm.Notify(EVENT.ENDBATTLE);
    }
    public override void Handle(BattleStateMachine bsm) ...
    public override void Exit(BattleStateMachine bsm) ...
}
```

```
public class DialogueController : Subject {

    private void NextDialogue()
    {
        if ((int)storyBoard[dialogueCount]["Number"] == -1) ...
        if (storyBoard[dialogueCount]["Talker"].ToString().Equals("전투"))
        {
            Notify(EVENT.GOBATTLE);
            dialogueCount++;
            return;
        }
```

전투 상태와 스토리 연출 상태 두 상태를 효율적으로 바꾸기 위해,
옵저버 패턴을 사용하였습니다.

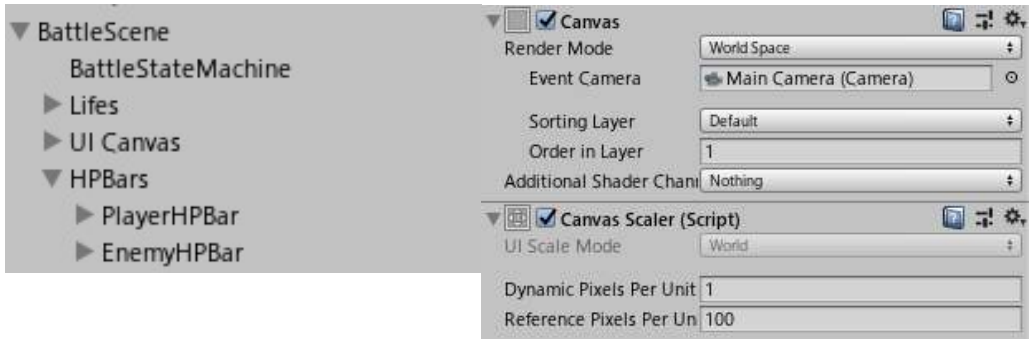
이를 통해, 전투와 스토리 연출을 오고 갈 때,
코드를 유지보수에 유리하도록 짤 수 있었습니다.

4. 문제 해결 과정 - HP바 위치 오류



HP바가 WorldToScreenPoint 함수를 통해
대상을 쫓아 다니도록 구현을 했는데.
해상도가 다르면 위치가 엉성하게 잡히고,
엉뚱한 값이 반환되어서 헤맸습니다.

4. 문제 해결 과정 - HP바 위치 오류




```
private void Update()
{
    UpdatePosition();
}

private void UpdatePosition()   // 타겟을 계속 추적
{
    transform.localPosition = thePivot.position;
}
```

HP바는 다른 캔버스에 담았고,
렌더 모드를 World Space로 해서
월드 좌표 계에서 계산되도록 하였습니다.

이렇게 설정하자, 해상도에 영향을 받지 않고
UI들이 잘 작동하는 것을 확인할 수 있었습니다.

게다가 WorldToScreenPoint함수가 불필요 해져서
빠르게 좌표계산을 할 수 있게 되었습니다.

여기까지 입니다.
감사합니다!

작성자 – 심교남