Smash Arena

- Unity 모바일 포트폴리오 -

포트폴리오 링크

https://github.com/Geonsu-Kim/SmashArena-Revision-

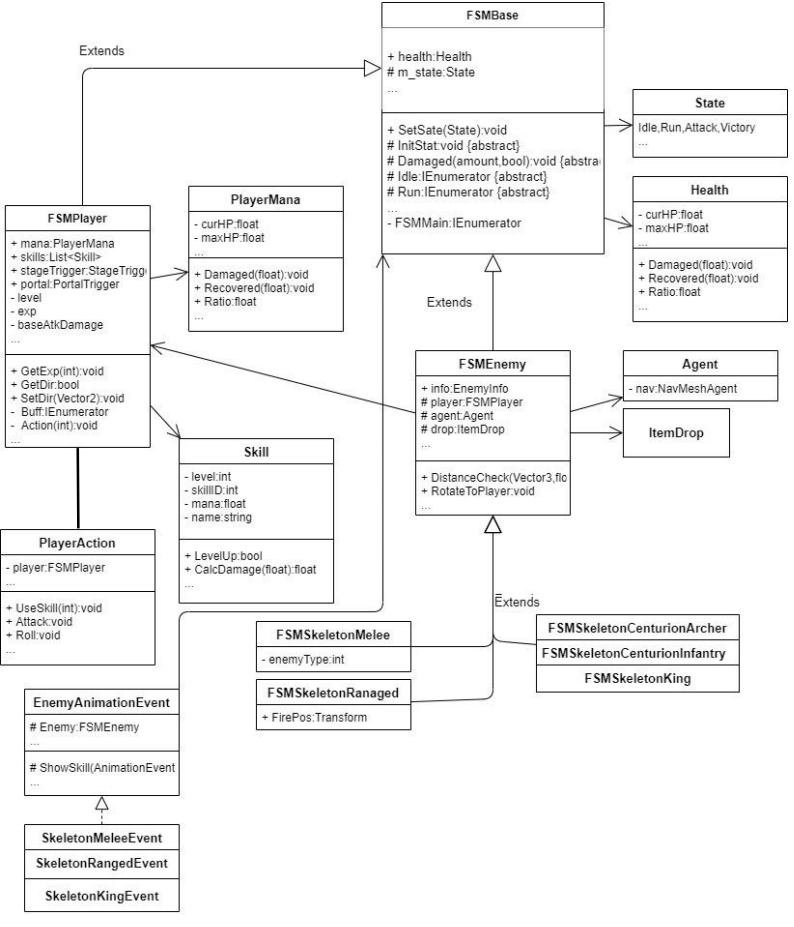
지원자 : 김건수

-Contents-

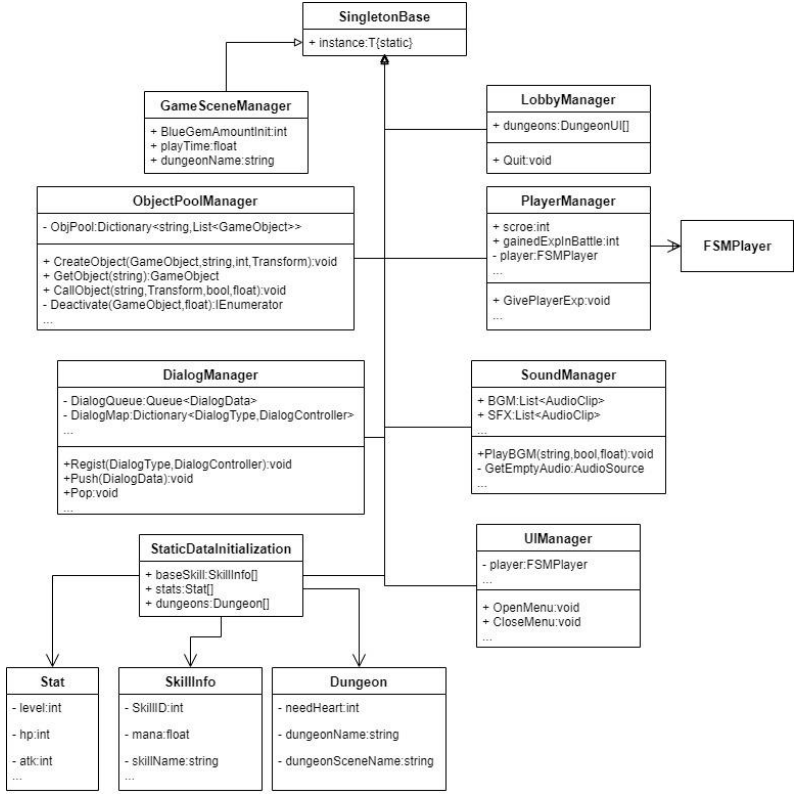
- 클래스 구조
- 디자인 패턴
- 데이터 관리
- 최적화 작업
- 그 외

클래스 구조도

캐릭터 클래스

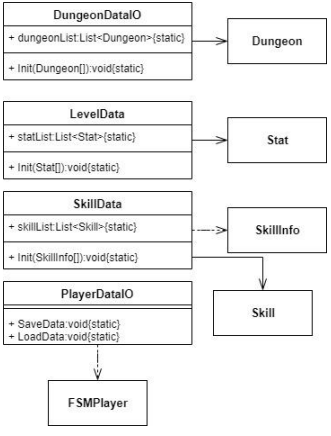


싱글턴 패턴 기반 클래스

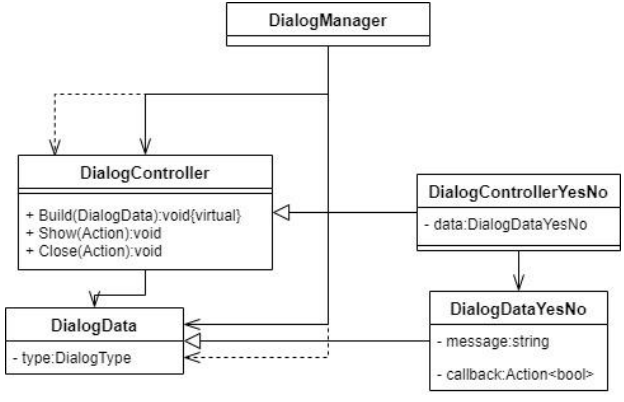


클래스 구조도

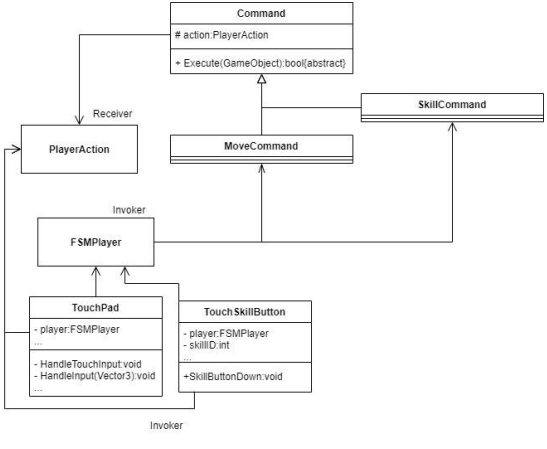
정적 데이터 관리 클래스



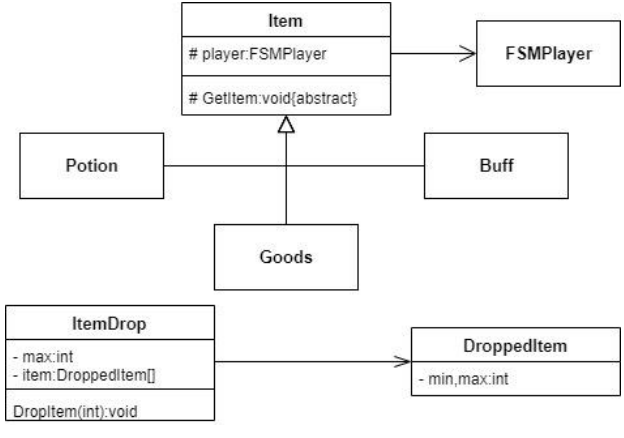
대화창 클래스



커맨드 클래스



아이템 클래스



디자인 패턴

싱글턴 패턴

```
public class SingletonBase<T> : MonoBehaviour where T : SingletonBase<T>
{
    public static T instance;
    참조 99+개
    public static T Instance
    {
        get
        {
            instance = GameObject.FindObjectOfType(typeof(T)) as T;
            return instance;
        }
    }
}
```

각 씬에서 음향 재생, UI 관리, 오브젝트 풀 등 **2개 이상의 인스턴스가 불필요한 기능**들은 싱글턴 패턴 기반 클래스를 상속받아 설계

활용 예시

```
sb.Length = 0;
sb.Append("PlayerGetItem");
SFXname = sb.ToString();
SoundManager.Instance.PlaySFX(SFXname);
```

사운드매니저를 통해 원하는 음향 재생

```
public void Spawn()
{
    if (target != null)
    {
        ObjectPoolManager.Instance.CallObject(target.name, this.transform);
    }
}
```

오브젝트 풀 매니저에서 스폰지점에서 정해진 몬스터 캐릭터를 소환

```
public class StaticDataInitialization : SingletonBase<StaticDataInitialization>
{
    // Start is called before the first frame update
    public SkillInfo[] baseSkill;
    public Stat[] stats;
    public Dungeon[] dungeons;
    ⚙ Unity 메시지 | 참조 0개
    void Awake()
    {
        SkillData.Init(baseSkill);
        LevelData.Init(stats);
        DungeonDataIO.Init(dungeons);
        SceneManager.LoadScene("scLobby");
    }
}
```

정적 데이터의 초기화는 단 한번만 실행하므로 싱글턴 패턴으로 설계

디자인 패턴

상태 패턴

```
참조 67개
public void SetState(State newState)
{
    isNewState = true;
    m_state = newState;
    animator.SetInteger(stringParam, (int)m_state);
}
참조 1개
IEnumerator FSMMain()
{
    while (true)
    {
        isNewState = false;
        if(health.IsDead())
            yield return StartCoroutine(Dead());
        yield return StartCoroutine(m_state.ToString());
    }
}


참조 7개
protected abstract IEnumerator Idle();
참조 7개
protected abstract IEnumerator Run();
참조 7개
protected abstract IEnumerator Attack();
참조 9개
protected abstract IEnumerator Dead();
```

FSM Base 클래스를 상속받아
플레이어와 적 캐릭터를 설계하고
m_state의 값(Idle, Run, Attack)에 따라 행동을 변경

추상 함수 구현

```
protected override IEnumerator Attack()
{
    agent.Stop();
    do
    {
        if (isDead()) break;
        yield return null;
        if (animator.GetCurrentAnimatorStateInfo(0).normalizedTime % 1.0f > 0.7f)
        {
            RotateToPlayer();
            if (enemyType==1)
            {
                animator.SetInteger("atkType", Random.Range(0, 2));
            }
            if (!DistanceCheck(player.transform.position, info.EnemyAtkRange))
            {
                SetState(State.Run);
            }
        }
    } while (!isNewState);
}
```

공격 애니메이션의 진행도가 70%이상일 경우
사정거리 내 플레이어의 존재 여부에 따라 행동 변경 여부 결정

디자인 패턴

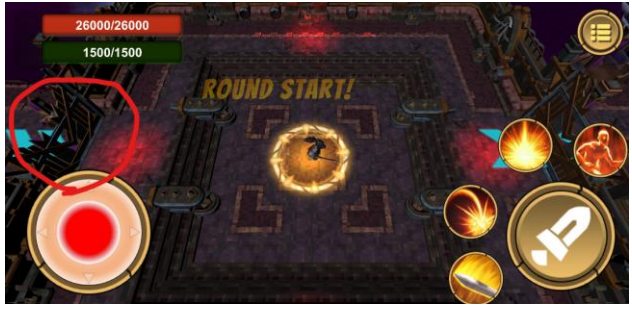
옵저버 패턴

```
public class Door : MonoBehaviour
{
    private Animator animator;


    public StageTrigger Trigger_Pre;
    public StageTrigger Trigger_Cur;
    Unity 메시지 | 참조 0개
    private void Awake()
    {
        animator = GetComponent<Animator>();
        if (Trigger_Pre != null)
        {
            Trigger_Pre.Doors_Next.Add(this);
        }
        if (Trigger_Cur != null)
        {
            Trigger_Cur.Doors_Cur.Add(this);
        }
    }
    참조 3개
    public void Open()
    {
        animator.SetBool("Open", true);

    }
    참조 2개
    public void Close()
    {

        animator.SetBool("Open", false);
    }
}
```

```
public class StageTrigger : MonoBehaviour
{
    [HideInInspector]  public List<Door> Doors_Cur;
    [HideInInspector]  public List<Door> Doors_Next;
```

```
public void StartBattle()
{
    PlayerManager.Instance.OnBattle = true;
    box.enabled = false;
    player.BtnNum = 0;
    ps.Stop();
    StartCoroutine(OnBattle());
    SoundManager.Instance.PlayBGM(bgmName);

    SoundManager.Instance.PlaySFX("DoorClose");
    if (BossEvent != null)
    {
        StartCoroutine(BossAppearence());
    }
    for (int i = 0; i < Doors_Cur.Count; i++)
    {
        Doors_Cur[i].Close();
    }
}
```

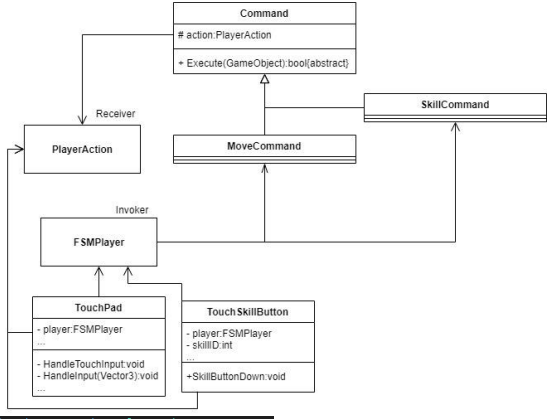
```
for (int i = 0; i < Doors_Cur.Count; i++)
{
    Doors_Cur[i].Open();
}
for (int i = 0; i < Doors_Next.Count; i++)
{
    Doors_Next[i].Open();
}
```



각 전투장에 배치된 출입문을 전장 트리거의
Door 리스트에 배치
전투 시작, 종료 시 전장 트리거에서 Door 리스트의
각 출입문에 개문, 폐문 호출을 전달함

디자인 패턴

커맨드 패턴

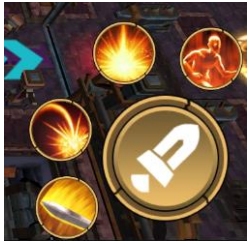



```
public abstract class Command
{
    protected PlayerAction action;

    참조 2개
    public Command(PlayerAction _action)
    {
        action = _action;
    }
    참조 3개
    public abstract bool Execute();
}
```

```
public class SkillCommand : Command
{
    private int m_num;
    참조 1개
    public SkillCommand(PlayerAction action,int num):base(action)
    {
        m_num = num;
    }
    참조 3개
    public override bool Execute()
    {
        if (action != null)
        {
            action.UseSkill(m_num);
            return true;
        }
        else return false;
    }
}
```

```
public class MoveCommand : Command
{
    참조 1개
    public MoveCommand(PlayerAction action) : base(action) { }
    참조 3개
    public override bool Execute()
    {
        if (action != null)
        {
            action.Move();
            return true;
        }
        return false;
    }
}
```

플레이어의 공격, 스킬 사용, 이동 등의 행동을 커맨드 패턴을 기반으로 구조화



클라이언트 : TouchPad(이동), TouchSkillButton(스킬 사용)

인보커 : FSMPlayer

리시버 : PlayerAction

디자인 패턴

커맨드 패턴 작동 순서

FSMPlayer는 번호에 따라 명령 목록을 가지고 있음

```
protected override void Awake()
{
    base.Awake();
    m_cc = GetComponent<CharacterController>();
    action = GetComponent<PlayerAction>();
    commandList = new Dictionary<int, Command>();
}
```

TouchPad와 TouchSkillButton(이하 컨트롤러)은 각자 moveCommand와 skillCommand를 생성하여 FSMPlayer에게 저장

```
private void Start()
{
    player = PlayerManager.Instance.Player;
    action = player.GetComponent<PlayerAction>();
    skillCommands = new SkillCommand(action, skill_ID);
    player.SetCommand(skill_ID, skillCommands);
```

```
public  void SkillButtonDown()
{
    if (skill_ID== 0)
    {
        player.ExecuteCommand(skill_ID);
    }
    else {
        if (PlayerManager.Instance.OnBattle)
        {
            if (!coolDown&&!player.IsUsingSkill())
            {
                coolDown = true;
                StartCoroutine(SkillCoolDown());
                player.ExecuteCommand(skill_ID);
            }
        }
```

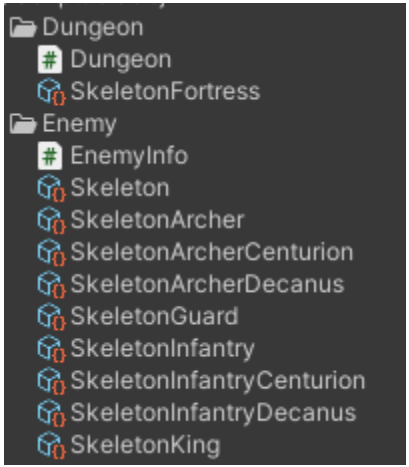
컨트롤러가 각자의 인덱스를 통해
플레이어(FSMPlayer)에게 행동 명령을 호출
FSMPlayer는 명령 목록에서 호출된 명령을 실행

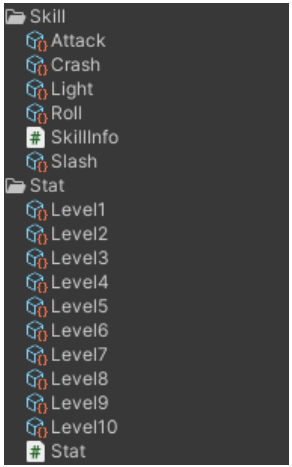
```
public void UseSkill(int num)
{
    switch (num)
    {
        case 0: Attack(); break;
        case 1: Roll(); break;
        case 2: Slash(); break;
        case 3: Crash(); break;
        case 4: Light(); break;

    }
}
참조 1개
public void Move()
{
    if (player.GetDir())
    {
        this.transform.rotation = Quaternion.LookRotation(player.Dir);
        player.m_cc.Move(player.Dir * Time.deltaTime * 2f);
        player.SetState(State.Run);
    }
}
```

PlayerAction은 FSMPlayer -> Command를 통해 전달받은 명령을 실행

데이터 관리

ScriptableObject





상수로 운용되는 정적 데이터들을 스크립터블 오브젝트로 만들어 인스턴스의 크기를 줄임

```
public class StaticDataInitialization : SingletonBase<StaticDataInitialization>
{
    public SkillInfo[] baseSkill;
    public Stat[] stats;
    public Dungeon[] dungeons;
    Unity 메시지 | 참조 0개
    void Awake()
    {
        SkillData.Init(baseSkill);
        LevelData.Init(stats);
        DungeonDataIO.Init(dungeons);
        SceneManager.LoadScene("scLobby");
    }
}
```

이 중 SkillInfo와 Stat, Dungeon은 클라이언트 실행 시 별도의 정적 클래스에 초기화, 저장하여 게임 도중 불러올 수 있도록 처리

예시로 플레이어의 레벨이 상승할 때 LevelData에서 해당 레벨의 정보를 가져와 갱신

데이터 관리

PlayerPrefs

```
public sealed class PlayerDataIO : MonoBehaviour
{
    참조 4개
    public static void SaveData()
    {
        FSMPlayer player = PlayerManager.Instance.Player;
        PlayerPrefs.SetInt("Level", player.Level);
        PlayerPrefs.SetInt("Exp", player.Exp);

    }
    참조 1개
    public static void LoadData()
    {
        FSMPlayer player = PlayerManager.Instance.Player;
        if (PlayerPrefs.HasKey("Level"))
        {
            player.Level = PlayerPrefs.GetInt("Level");
            player.Exp = PlayerPrefs.GetInt("Exp");
        }
        else
        {
            player.Level = 1;
            player.Exp = 0;
        }

    }
}
```

플레이어의 레벨이나 보유 경험치와 같이 자주 변하는
데이터는 PlayerPrefs를 통해 레지스트리에 저장

데이터 관리

XML

```
public sealed class PlayerDataIO : MonoBehaviour
{
    public static void SaveData()
    {
        FSMPlayer player = PlayerManager.Instance.Player;

        XmlDocument Parent = new XmlDocument();
        XmlElement PlayerNode = Parent.CreateElement("PlayerDB");
        Parent.AppendChild(PlayerNode);

        XmlElement PlayerStatNode = Parent.CreateElement("Player");

        PlayerStatNode.SetAttribute("Level", player.Level.ToString());
        PlayerStatNode.SetAttribute("Exp", player.Exp.ToString());

        PlayerNode.AppendChild(PlayerStatNode);
        Parent.Save(Application.dataPath + "/Data/PlayerData.xml");
    }
    public static void LoadData()
    {
        if (!System.IO.File.Exists(Application.dataPath + "/Data/PlayerData.xml")) return;

        FSMPlayer player = PlayerManager.Instance.Player;
        XmlDocument Parent = new XmlDocument();
        Parent.Load(Application.dataPath + "/Data/PlayerData.xml");
        XmlElement PlayerNode = Parent["PlayerDB"];
        XmlNodeList playerStats = PlayerNode.ChildNodes;
        foreach(XmlElement stats in PlayerNode.ChildNodes)
        {
            player.Level = System.Convert.ToInt32(stats.GetAttribute("Level"));
            player.Exp = System.Convert.ToInt32(stats.GetAttribute("Exp"));

        }
    }
}
```

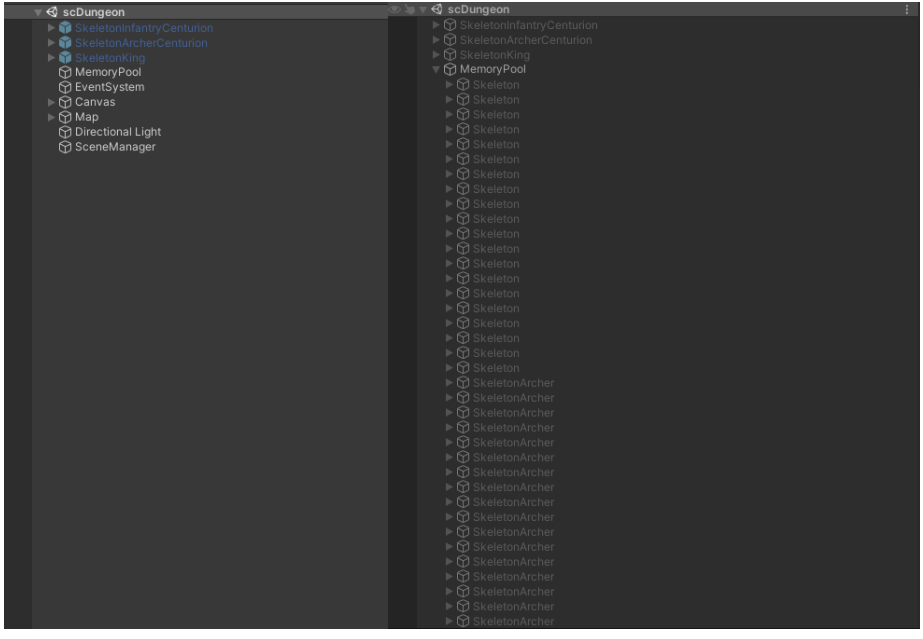
개발 초기에는 XML 파싱을 이용하여 플레이어 정보와
정적 데이터를 관리하려고 했으나 모바일의 저장 경로를
찾지 못하여 데이터 로드에 실패하는 경우가 자주 발생
이에 스크립터블 오브젝트와 PlayerPref로 관리 방식 전환

문제 해결을 위해 사용한 모바일 저장 경로 양식
Resources.Load()->메모리 적재 문제 및 로드 실패

"jar:file://" + Application.dataPath + "!/assets“
-> StreamingAsset 폴더에 XML파일을 저장하여
빌드 시 해당 앱 경로에 XML파일을 함께 저장함
로드 실패 및 읽기만 가능하여 플레이어 정보 갱신 불가

최적화 작업

오브젝트 풀링



씬 실행 시 초기에 미리 객체를 할당한 후
Active On/Off로 재사용하여 객체의 생성, 삭제에서
발생하는 GC를 차단

최적화 작업

오브젝트 풀링

```
public class ObjectPoolManager : SingletonBase<ObjectPoolManager>
{


    [SerializeField] private GameObject[] Prefabs;
    private List<GameObject> ObjPool;
```

```
public GameObject GetObject(string name)
{
    if (ObjPool.Count == 0) return null;
    for (int i = 0; i < ObjPool.Count; i++)
    {
        if (i == ObjPool.Count - 1)
        {
            CreateObject(name, 1);
            return ObjPool[i + 1];
        }
        if (!ObjPool[i].name.Equals(name)) continue;
        if (!ObjPool[i].activeSelf)return ObjPool[i];
    }
    return null;
}
```

최적화 작업 이전에는 List 타입의 풀에 모든 객체를 저장

오브젝트 풀에서 원하는 객체를 찾는 과정에서 대량의 GC 발생

아래는 플레이어가 다수의 적 캐릭터를 공격하는 순간의 프레임

▼ PlayerLoop	87.9%	0.2%	2	0.7 MB	32.13	0.08
▼ PreLateUpdate.DirectorUpdateAnimationBegin	70.9%	0.0%	1	0.7 MB	25.91	0.02
▼ Director.ProcessFrame	70.6%	0.0%	1	0.7 MB	25.81	0.00
▼ Animators.Update	70.6%	0.0%	1	0.7 MB	25.81	0.00
▼ Animators.FireAnimationEventsAndBehaviours	70.1%	0.0%	1	0.7 MB	25.62	0.00
▼ Animator.FireAnimationEvents	70.1%	0.0%	17	0.7 MB	25.61	0.02
▼ PlayerAnimationEvent.OnSkill1()	70.0%	49.4%	1	0.7 MB	25.59	18.06
▶ GameObject.Activate	6.6%	0.0%	58	12.5 KB	2.42	0.03
▶ Mono.JIT	5.4%	5.4%	18	330 B	1.98	1.98
FindObjectsOfType	3.4%	3.4%	93	0 B	1.27	1.27
GC.Alloc	1.7%	1.7%	14102	0.6 MB	0.62	0.62

최적화 작업

오브젝트 풀링

```
public class ObjectPoolManager : SingletonBase<ObjectPoolManager>
{


    [SerializeField] private GameObject[] Prefabs;
    private Dictionary<string, List<GameObject>> ObjPool;
```

```
public GameObject GetObject(string name)
{
    if (ObjPool[name].Count == 0) return null;
    for (int i = 0; i < ObjPool[name].Count; i++)
    {
        if (i == ObjPool[name].Count - 1)
        {
            CreateObject(name, 1);
            return ObjPool[name][i];
        }
        if (!ObjPool[name][i].activeSelf) return ObjPool[name][i];
    }
    return null;
}
```

객체의 이름을 키로 하는 Dictionary 타입으로 변경한 후
GC 호출이 상당히 감소함

▼ PlayerLoop	89.3%	0.2%	2	45.8 KB	24.97	0.07
▼ PreLateUpdate.DirectorUpdateAnimationBegin	70.7%	0.0%	1	22.6 KB	19.77	0.02
▼ Director.ProcessFrame	70.3%	0.0%	1	22.6 KB	19.67	0.00
▼ Animators.Update	70.3%	0.0%	1	22.6 KB	19.67	0.00
▼ Animators.FireAnimationEventsAndBehaviours	69.6%	0.0%	1	22.6 KB	19.48	0.00
▼ Animator.FireAnimationEvents	69.6%	0.1%	17	22.6 KB	19.48	0.02
▼ PlayerAnimationEvent.OnSkill1()	69.5%	44.4%	1	22.6 KB	19.45	12.42
▶ GameObject.Activate	7.7%	0.0%	57	12.5 KB	2.16	0.02
▶ Mono.JIT	6.9%	6.9%	18	330 B	1.94	1.94
FindObjectsOfType	4.5%	4.5%	92	0 B	1.26	1.26
▶ PlayerAnimationEvent.Shake() [Coroutine: MoveNext]	1.6%	0.1%	1	0 B	0.45	0.03
▶ GameObject.Deactivate	1.4%	0.0%	17	192 B	0.41	0.00
▶ ObjectPoolManager.Deactivate() [Coroutine: MoveNext]	1.0%	0.1%	17	300 B	0.29	0.03
▶ Instantiate	0.8%	0.0%	1	0 B	0.24	0.00
▶ FSMPlayer.ColorByHit() [Coroutine: MoveNext]	0.4%	0.2%	16	0 B	0.13	0.07
▶ ObjectPoolManager.Deactivate() [Coroutine: System.Collections.IEnumerator.get_Current]	0.0%	0.0%	17	0 B	0.02	0.00
▶ PlayerAnimationEvent.Shake() [Coroutine: System.Collections.IEnumerator.get_Current]	0.0%	0.0%	1	0 B	0.02	0.00
GC.Alloc	0.0%	0.0%	266	9.3 KB	0.02	0.02

최적화 작업

문자열 처리

임시 문자열의 생성을 줄이고자 stringBuilder와 compare Tag()활용

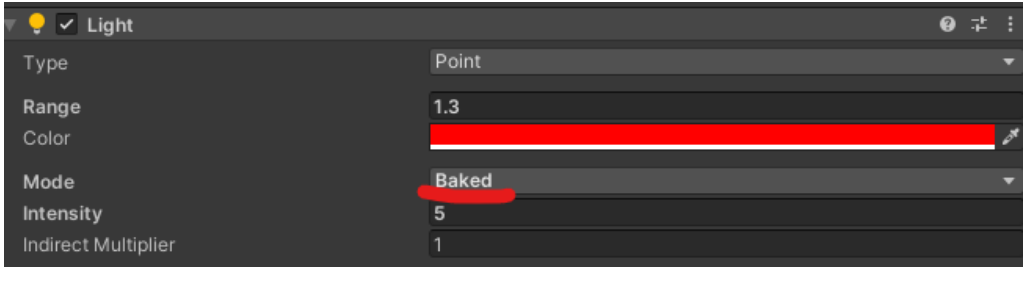
```
public void OnSkill1()
{
    stringBuilder.Length = 0;
    stringBuilder.Append(playerName);
    stringBuilder.Append("Slash");
    SFXname = stringBuilder.ToString();
    SoundManager.Instance.PlaySFX(SFXname);
```

```
protected override void OnTriggerEnter(Collider other)
{
    if ((this.gameObject.CompareTag("PlayerAttack") && other.gameObject.CompareTag("Enemy"))
    || (this.gameObject.CompareTag("EnemyAttack") && other.gameObject.CompareTag("Player")))
    {
```

최적화 작업

라이트 매핑

적 캐릭터 스폰 지점을 표시하는 광원은 정적 광원이며, 플레이어나 적 캐릭터와 같은 동적 객체와 상호 연관성을 부여할 필요가 없어 라이트 매핑 처리



그 외

적 캐릭터의 스킬 사용

```
protected override void OnEnable()
{
    base.OnEnable();
    skillCooldown = false;
    SkillQueue.Clear();
    for (int i = 0; i < 4; i++)
    {
        StartCoroutine(SkillCoolDown(i));
    }
    StartCoroutine(QueueCoolDown(5f));
}
//스킬 자체의 쿨타임(쿨타임이 차면 스킬 준비 큐에 삽입된다)
참조 5개
private IEnumerator SkillCoolDown(int num)
{

    yield return YieldInstructionCache.WaitForSeconds(MaxCooltime[num]);
    switch (num)
    {
        case 0: SkillQueue.Enqueue(State.Skill1); break;
        case 1: SkillQueue.Enqueue(State.Skill2); break;
        case 2: SkillQueue.Enqueue(State.Skill3); break;
        case 3: SkillQueue.Enqueue(State.Skill4); break;
    }
}
```

```
protected override IEnumerator Run()
{
    do
    {
        if (isDead()) break;
        agent.TraceTarget(Player.transform.position);
        yield return null;
        if (DistanceCheck(player.transform.position, info.EnemyChasingRange))
        {
            if (SkillQueue.Count != 0 && skillCooldown)
            {
                agent.Stop();
                StartCoroutine(QueueCoolDown(10f));
                SetState(SkillQueue.Dequeue());
                break;
            }
            if (DistanceCheck(player.transform.position, info.EnemyAtkRange))
            {
                SetState(State.Attack);
            }

        }
        else
        {
            SetState(State.Idle);
        }
    } while (!isNewState);
}
참조 7개
```

하나의 스킬 큐에 보유한 스킬들을 푸쉬하고 쿨타임이
종료됐을 때 제일 앞의 스킬을 팝하여 스킬 사용 명령 실행

그 외

데미지 폰트

TextMeshProUGUI 사용

카메라를 바라보도록 렌더링하고 별도의 애니메이션을
추가하여 데미지 폰트 생성 시 폰트가 위로 올라가며 서서히
사라지도록 구현
생성은 오브젝트 풀링으로 구현

Thank ou For Watching