소스 코드 리뷰

길용현 – 아비규환 제작 코드 리뷰

https://youtu.be/SoN3IVONKro

https://github.com/Gilssom/Aura2_Test_Project.git

22_10_29 제작

목차 및 개요

캐릭터 구현 관련

1-1. 캐릭터 능력치

캐릭터 상태 능력치에 관련해서는
객체를 따로 참조하지 않고 메서드만 참조할 수 있는
Delegate 문법을 활용해
CallBack 함수를 구현하여 사용하였습니다.

클래스의 변수는 private 로 감추면서도
추가적인 논리가 필요하지 않고 접근할 수 있게
프로퍼티를 설정하였습니다.

캐릭터 능력치 (PlayerStats.cs) 는
싱글톤이 구현되어 있어
필요에 따른 함수와 프로퍼티에 접근해
실시간 전투 진행 상황에 대한 플레이어 능력치를
조절할 수 있습니다.

```
public delegate void OnHealthChangedDelegate();
public OnHealthChangedDelegate onHealthChangedCallback;
```

```
void ClampHealth()
{
    health = Mathf.Clamp(health, 0, maxHealth);

    if (onHealthChangedCallback != null)
        onHealthChangedCallback.Invoke();
}
```

```
[SerializeField]
private float health;
[SerializeField]
private float maxHealth;
[SerializeField]
private float maxTotalHealth;
[SerializeField]
private float soul;
[SerializeField]
private float maxSlashGage;
[SerializeField]
private float SlashGage;

public float Health { get { return health; } }
public float MaxHealth { get { return maxHealth; } }
public float MaxTotalHealth { get { return maxTotalHealth; } }
public float Soul { get { return soul; } }
public float Slash { get { return SlashGage; } }
public float MaxSlash { get { return maxSlashGage; } }
```

캐릭터 구현 관련

1-2. 캐릭터 이동 및 회피

캐릭터의 기본 이동은
유니티의 Input System 의
방향키에 따른 값을 가져오고

대각선 이동에도 동일한 속도를
제공하기 위해 정규화를 하였습니다.

졸업 작품을 진행하면서
애니메이션과 사운드 리소스가
서로 타이밍이 맞지 않고
플레이에 어색한 부분이 있었습니다.

그래서 간단히 해결하기 위해
따로 isMoving 변수를 추가하여
발자국 사운드를 관리하였습니다.

스페이스바의 회피기를 사용할 때
플레이어의 이동속도를 조절하여
구현하였습니다.

```
void GetInput()
{
    HAxis = Input.GetAxis("Horizontal");
    VAxis = Input.GetAxis("Vertical");
}

void Move()
{
    bool isMoving = false;

    moveVec = new Vector3(HAxis, 0, VAxis).normalized;

    transform.position += moveVec * Speed * Time.deltaTime;

    transform.LookAt(transform.position + moveVec);

    m_Animator.SetBool("isRun", moveVec != Vector3.zero);

    if (moveVec != Vector3.zero && !isDodge)
        isMoving = true;
    else
        isMoving = false;

    if (isMoving)
    {
        if (!m_FootSFX.isPlaying)
            m_FootSFX.Play();
    }
    else
        m_FootSFX.Stop();
}
```

```
IEnumerator Dodge()
{
    if (moveVec != Vector3.zero && !isDodge) // ! = bool 형태의 반대(=false)
    {
        Speed *= 2;
        m_Animator.SetTrigger("DoJump");
        SoundManager.Instance.SFXPlay("Charge Attack", m_clip[4]);
        isDodge = true;

        yield return new WaitForSeconds(0.7f);

        Speed *= 0.5f;
        isDodge = false;
    }
}
```

캐릭터 구현 관련

1-3. 캐릭터 공격 시스템

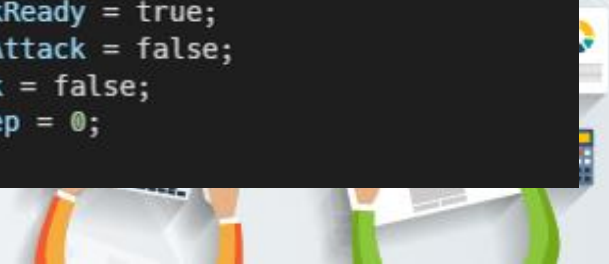
캐릭터의 기본 공격은
게임 특성 상 쿼터뷰이기 때문에
ScreenPointToRay 로
마우스의 클릭 좌표를 가져와
해당 방향을 바라보며 공격하는
시스템으로 구현하였습니다.

기본 공격을 진행하면
ComboStep 이 1 씩 쌓이게 되고
애니메이션 이벤트를 활용해
공격 도중 다음 공격을 실행하는
마우스를 클릭하지 않으면
ComboReset 을 호출해
초기화를 시키는 방식을 활용했습니다.

3번째 공격 이후는 자동으로
초기화를 시키게 하였습니다.

```
void Attack()
{
    if (Input.GetMouseButtonDown(0) && !isAttack)
    {
        if (!EventSystem.current.IsPointerOverGameObject())
        {
            Ray ray = m_Camera.ScreenPointToRay(Input.mousePosition);
            RaycastHit rayhit;
            if (Physics.Raycast(ray, out rayhit, 100))
            {
                Vector3 nextVec = rayhit.point - transform.position;
                nextVec.y = 0;
                transform.LookAt(transform.position + nextVec);
            }

            m_FootSFX.Stop();
        }
    }

    if (Input.GetMouseButtonDown(0) && comboStep == 0)
    {
        if(!EventSystem.current.IsPointerOverGameObject())
        {
            m_Animator.SetBool("isRun", false);

            m_Animator.SetTrigger("Attack");
            comboStep = 1;
            isAttackReady = false;
            m_FinalAttack = true;
            return;
        }
    }
```

```
if (comboStep != 0)
{
    if (ComboPossible && Input.GetMouseButtonDown(0))
    {
        m_FinalAttack = true;
        ComboPossible = false;
        comboStep += 1;
    }
}
void Combopossible()
{
    ComboPossible = true;
}
void Combo()
{
    if (comboStep == 2)
    {
        m_Animator.SetTrigger("SecondAttack");
    }

    else if (comboStep == 3)
    {
        m_Animator.SetTrigger("FinalAttack");
    }
}

void ComboReset()
{
    m_Animator.ResetTrigger("SecondAttack");
    m_Animator.ResetTrigger("FinalAttack");
    ComboPossible = false;
    isAttackReady = true;
    m_FinalAttack = false;
    isAttack = false;
    comboStep = 0;
}
```



캐릭터 구현 관련

1-4. 주변 탐색

Linq 를 이용하여 주변 오브젝트 탐색을
구현하였습니다.

FindNearObjByTag 의 string tag 에
입력된 태그에 맞는 오브젝트 목록을
List 에 저장을 먼저 하였습니다.

그 후, Linq. OrderBy 를 활용해
캐릭터와 List 에 저장된 오브젝트들 간의
거리를 계산해 정렬을 한 다음

FirstOrDefault 를 활용해
그 정렬된 값의 첫번째 인자를 가져옵니다.
없으면 Default 를 반환합니다.

따라서 m_NearNpc 에 추가를 한 다음
플레이어에서 참조할 수 있도록 합니다.

```
using System.Linq;

public class NearNpcCheck : MonoBehaviour
{
    private ChoheeController m_Player;

    [SerializeField]
    public GameObject m_NearNpc;

    public float m_ItemCheckDis;

    private GameObject FindNearObjByTag(string tag)
    {
        var objects = GameObject.FindGameObjectsWithTag(tag).ToList();

        var neareastObject = objects
            .OrderBy(obj =>
            {
                return Vector3.Distance(transform.position, obj.transform.position);
            })
        .FirstOrDefault();

        m_NearNpc = neareastObject;

        return neareastObject;
    }
```

몬스터 구현 관련

2-1. 몬스터 능력치

열거형 enum 타입으로
각각의 몬스터를 정의하고
Switch – case 문을 같이 사용해
각각의 능력치를 부여하였습니다.

m_TraceDis 는 해당 거리 안에
플레이어가 있으면 Trace 가
될 수 있도록 하는 값이므로
일반 지역에서의 몬스터는 짧게,
스폰되는 몬스터들은 넓게 하였습니다.

```
switch (m_EnemyType)
{
    case EnemyType.HellGhost:
        if (gameObject.name == "GhostMonster_NormalMask")
            m_HaveMaskNum = 1;
        else if (gameObject.name == "GhostMonster_SpeedMask")
        {
            m_HaveMaskNum = 2;
            m_MaxHP = 750;
        }
        else if (gameObject.name == "GhostMonster_FireMask")
        {
            m_HaveMaskNum = 3;
            m_MaxHP = 1000;
        }
```

```
switch (m_EnemyType)
{
    case EnemyType.HellGhost:
        if (isSpawnMonster)
            m_TraceDis = 100;
        else
            m_TraceDis = 10;

        m_AttackDis = 3;
        m_rotSpeed = 3;
        m_moveSpeed = 2;
        m_MaxHP = 600;

        m_MinDrop = 1;
        m_MaxDrop = 3;
        break;
    case EnemyType.FireMonster:
        if (isSpawnMonster)
            m_TraceDis = 100;
        else
            m_TraceDis = 7;

        m_AttackDis = 2;
        m_rotSpeed = 3;
```

몬스터 구현 관련

2-2. 몬스터 상태변환

일정 시간 마다 상태변환 코루틴을 호출합니다.
각각의 몬스터들은 플레이어 감지 범위와
공격 범위를 가지고 있습니다.

플레이어가 감지 범위 내에 들어오면
Trace 를 실행하며 추적하게 됩니다.

플레이어가 공격 범위 내에 들어오면
Attack 을 실행하며 공격을 하게 됩니다.

그 외의 상황에서는 기본 동작을 취합니다.

도깨비불 몬스터는 랜덤한 시간 마다
자신의 주변 랜덤한 위치로 움직이게 하여
시각적인 효과를 주었습니다.

```
IEnumerator CheckStateForAction()
{
    while(!isDeath)
    {
        switch (m_CurState)
        {
            case CurState.idle:
                if (m_EnemyType != EnemyType.FireMonster)
                    m_Anim.SetBool("isRun", false);
                m_Anim.SetBool("isAttack", false);

                while (m_CurState == CurState.idle && !isSpawnMonster && m_EnemyType ==
                    EnemyType.FireMonster)
                {
                    Vector3 curPos = this.transform.position;
                    float dir1 = Random.Range(-0.3f, 0.3f);
                    float dir2 = Random.Range(-0.3f, 0.3f);
                    float ranTime = Random.Range(0, 3);

                    yield return new WaitForSeconds(ranTime);
                    this.transform.DOMove(new Vector3(curPos.x + dir1, curPos.y, curPos.z + dir2),
                     1);
                }
                break;
            case CurState.trace:
                m_NavAgent.speed = m_moveSpeed;
                m_Rigid.isKinematic = true;
                m_NavAgent.destination = m_Player.transform.position;
                m_NavAgent.Resume();
                LookPlayer();
                m_Anim.SetBool("isAttack", false);
                if (m_EnemyType != EnemyType.FireMonster)
                    m_Anim.SetBool("isRun", true);
                break;
            case CurState.attack:
                isAttack = true;
                m_NavAgent.speed = 0;
                m_Rigid.isKinematic = false;
                LookPlayer();
                if (m_EnemyType != EnemyType.FireMonster)
                    m_Anim.SetBool("isRun", false);
                m_Anim.SetBool("isAttack", true);
                break;
            case CurState.death:
                break;
            default:
                break;
```

몬스터 구현 관련

2-3. 몬스터 공격

일정 시간마다 상태 체크하는 코루틴을
반복적으로 호출해 행동합니다.

열거형 enum 타입으로
공격 , 추적 , 기본 상태를 설정하였고

플레이어와의 거리를 코루틴 호출 시
체크해 설정해둔 거리보다 가까우면
추적 > 공격 사거리 체크 > 공격 을 합니다.

랜턴 몬스터에서 나오는 불꽃 발사체는
오브젝트 풀링 시스템을 사용해
공격 시 객체를 활성화 시킨 다음
사용할 수 있게 하였습니다.

```
IEnumerator CheckState()
{
    while(!isDeath)
    {
        yield return new WaitForSeconds(0.3f);

        float Dis = Vector3.Distance(m_Player.transform.position, this.transform.position);
        if (Dis <= m_AttackDis)
            m_CurState = CurState.attack;
        else if (Dis <= m_TraceDis)
            m_CurState = CurState.trace;
        else if(m_TraceDis < Dis)
            m_CurState = CurState.idle;
    }
}
```

```
case CurState.attack:
    isAttack = true;
    m_NavAgent.speed = 0;
    m_Rigid.isKinematic = false;
    LookPlayer();
    if (m_EnemyType != EnemyType.FireMonster)
        m_Anim.SetBool("isRun", false);
    m_Anim.SetBool("isAttack", true);
    break;
```

```
void FireBallAttack()
{
    GameObject FireBall = ObjectPoolManager.instance.m_ObjectPoolList[0].Dequeue();
    FireBall.SetActive(true);

    FireBall.transform.position = m_FirePos.position;
    FireBall.transform.rotation = m_FirePos.rotation;
    Rigidbody BallRigid = FireBall.GetComponent<Rigidbody>();
    BallRigid.velocity = m_FirePos.forward * 20;
}
```

몬스터 구현 관련

2-4. 사망 이벤트

몬스터는 사망 시 디졸브 효과를 재생하며
재생이 끝나면 아이템을 드랍합니다.

각자 변수에 해당되는 최소 , 최댓값 사이
몇 개를 드랍 할 건지 설정하고
랜덤한 주변 위치값을 설정합니다.

드랍아이템 또한 오브젝트 풀링을 활용해
객체 주변에 활성화가 되도록 한 다음
For 문으로 해당 개수 만큼 실행합니다.

해당 몬스터가 특정 아이템을 갖고 있는
설정이라면 해당 아이템을 플레이어가
착용 할 수 있도록 드랍합니다.

```
void DeathEvent()
{
    curPos = this.transform.position;

    if (Cutoff >= MaxCutoff)
    {
        Destroy(this.gameObject);
        GameManager.Instance.m_KillCount += 1;

        int ItemCount = Random.Range(m_MinDrop, m_MaxDrop);
        for (int i = 0; i < ItemCount; i++)
        {
            float RandomTF = Random.Range(0.5f, 3);
            GameObject SoulItem = ObjectPoolManager.instance.m_ObjectPoolList[6].Dequeue();
            SoulItem.SetActive(true);
            SoulItem.transform.position = new Vector3(curPos.x + RandomTF, curPos.y + 0.5f,
                curPos.z + RandomTF);
            SoulItem.transform.rotation = Quaternion.identity;
        }

        int HealthDrop = Random.Range(0, 100);
        if (HealthDrop <= 10 && PlayerStats.Instance.Health < PlayerStats.Instance.MaxHealth)
        {
            GameObject HealItem = ObjectPoolManager.instance.m_ObjectPoolList[7].Dequeue();
            HealItem.SetActive(true);
            HealItem.transform.position = new Vector3(curPos.x, curPos.y + 0.5f, curPos.z);
            HealItem.transform.rotation = Quaternion.identity;
        }

        if (m_HaveMaskNum != 0)
        {
            string ItemName = m_DropMask[m_HaveMaskNum].name;

            GameObject Mask = Instantiate(m_DropMask[m_HaveMaskNum],
                new Vector3(curPos.x, curPos.y + 0.2f, curPos.z), Quaternion.Euler(0, 0, 180));
            Mask.name = ItemName;
        }
    }
}
```

보스 구현 관련

3-1. 보스 능력치

보스 몬스터는 총 3 페이즈로
구성이 되어 있습니다.

게임 기획서에 설정된 값에 따라
체력에 따른 페이즈를 나뉘었습니다.

체력에 따른 보스 체력 UI 에
부드러운 움직임을 주기 위해서
선형보간 함수를 사용하였습니다.

```
void HealthCtrl()
{
    if (m_CurHealth > 4200 && m_CurHealth < 9100)
        m_PhaseNumber = 2;
    else if (m_CurHealth <= 4200)
        m_PhaseNumber = 3;
    if (m_CurHealth <= 0)
        Death();

    m_HpBar.value = Mathf.Lerp(m_HpBar.value, m_CurHealth, Time.deltaTime * 3f);

    float HpPercent = m_CurHealth / 120;
    m_HpPercent.text = HpPercent.ToString("F0") + "%";
}
```

* 인게임 보스 UI HP Bar

보스 구현 관련

3-2. 보스 공격 설정

보스는 각 페이즈 별 랜덤 공격 패턴이 존재합니다.

매 공격이 끝나면 다시 코루틴 함수를 호출하여
다음 턴에 어떤 공격을 할 지 결정합니다.

게임 기획서에 설정된 공격 확률에 따라
Random.Range 로 설정을 하였고

일반 공격 또한 4가지의 방향으로 공격을 하기 때문에
각 방향을 25% 확률로 설정하였습니다.

```
public IEnumerator BossThink()
{
    int OnePattenAttack = Random.Range(0, 2);
    int TwoPattenAttack = Random.Range(0, 3);

    yield return new WaitForSeconds(2);

    if(m_PhaseNumber == 1)
    {
        switch (OnePattenAttack)
        {
            case 0:
                BossNormalAttack();
                Debug.Log("페이즈 1 : 일반공격");
                break;
            case 1:
                BossRotateAttack();
                Debug.Log("페이즈 1 : 회전공격");
                break;
        }
    }

    else if (m_PhaseNumber == 2)
    {
        switch (TwoPattenAttack)
        {
            case 0:
                BossNormalAttack();
                Debug.Log("페이즈 2 : 일반공격");
                break;
            case 1:
                BossRotateAttack();
                Debug.Log("페이즈 2 : 회전공격");
                break;
            case 2:
                BossMovingAttack();
                Debug.Log("페이즈 2 : 이동공격");
                break;
        }
    }

    else if (m_PhaseNumber == 3)
    {
        switch (TwoPattenAttack)
```

보스 구현 관련

3-3. 보스 페이즈 별 공격 패턴

3페이즈에서 실행되는 폭탄 공격입니다.

애니메이션의 길이에 따라 생성되는
폭탄의 개수를 조절하였습니다.

For문으로 떨어질 폭탄의 개수만큼
객체 주변의 랜덤한 좌표를 가져와
오브젝트 풀링으로 해당 좌표에
활성화를 시키고 폭탄에
RigidBody 값을 따로 설정해
낙하 시키도록 하였습니다.

이동 공격은 PathCreator 플러그인을
따로 설치해 이동 경로를 그려서
구현하였습니다.

```
IEnumerator BossLongDisAttack()
{
    SoundManager.Instance.SFXPlay("Long Attack", m_SoundEffect[2]);
    m_Anim.SetBool("Long Attack", true);
    Vector3 Pos = this.transform.position;
    int AttackNumbers = Random.Range(10, 15);

    for (int i = 0; i < AttackNumbers; i++)
    {
        int RandomColor = Random.Range(2, 6);
        float RandomTFX = Random.Range(-10f, 10f);
        float RandomTFZ = Random.Range(-10f, 10f);

        GameObject BombObject = ObjectPoolManager.instance.m_ObjectPoolList[RandomColor].Dequeue();

        BombObject.SetActive(true);

        BombObject.transform.position = new Vector3(Pos.x + RandomTFX, Pos.y + 20, Pos.z +
            RandomTFZ);
        BombObject.transform.rotation = Quaternion.identity;

        yield return new WaitForSeconds(0.3f);
    }

    yield return new WaitForSeconds(1);
    m_Anim.SetBool("Long Attack", false);
    StartCoroutine(BossThink());
}
```

```
void BossMovingAttack()
{
    m_Anim.SetBool("Right Tornado Attack", true);
    m_TornadoArea.enabled = true;
    m_PathSystem.enabled = true;
    m_MovingLight.SetActive(true);
}
```

보스 구현 관련

3-4. 보스 사망 이벤트

보스도 몬스터와 동일하게 사망 시 디졸브 효과를 재생하며 재생이 끝나면 아이템을 드랍합니다.

게임 기획서에 설정되어 있는 아이템 드랍 개수를 랜덤으로 설정하고

해당 값만큼 for문을 실행시켜 활성화 시킵니다.
동일하게 오브젝트 풀링을 활용하였습니다.

```
void Death()
{
    if (!isDeath)
        SoundManager.Instance.SFXPlay("Death", m_SoundEffect[6]);
    isDeath = true;
    m_HpBar.gameObject.SetActive(false);
    CancelInvoke();
    StopAllCoroutines();
    m_Anim.Play("Death");

    Vector3 Pos = this.transform.position;

    CapsuleCollider Bosscoll = GetComponent<CapsuleCollider>();

    Bosscoll.enabled = false;
    m_Rigid.isKinematic = true;

    if (Cutoff >= MaxCutoff)
    {
        GameManager.Instance.m_FirstBossEnd = true;
        Destroy(this.gameObject);
        GameManager.Instance.m_KillCount += 1;

        int MaxItems = Random.Range(20, 40);
        for (int i = 0; i < MaxItems; i++)
        {
            float RandomTFX = Random.Range(-8, 8);
            float RandomTFZ = Random.Range(-8, 8);

            GameObject SoulItem = ObjectPoolManager.instance.m_ObjectPoolList[6].Dequeue();
            SoulItem.SetActive(true);
            SoulItem.transform.position = new Vector3(Pos.x + RandomTFX, Pos.y + 0.5f, Pos.z +
                RandomTFZ);
            SoulItem.transform.rotation = Quaternion.identity;
        }

        int HealthDrop = Random.Range(0, 100);
        if (HealthDrop <= 100 && PlayerStats.Instance.Health < PlayerStats.Instance.MaxHealth)
        {
            GameObject HealItem = ObjectPoolManager.instance.m_ObjectPoolList[7].Dequeue();
            HealItem.SetActive(true);
            HealItem.transform.position = new Vector3(Pos.x, Pos.y + 0.5f, Pos.z);
            HealItem.transform.rotation = Quaternion.identity;
        }
    }

    Cutoff += Speed;
    if (Cutoff != MaxCutoff)
    {
        m SkinmeshRenderer.material.SetFloat(" Dissolve", Cutoff);
```

게임 시스템 관련

4-1. 게임 진행 시스템

플레이어는 드랍되는 아이템에 설정되어 있는
Type 에 따라 스탯에 영향을 주게 하였습니다.

PlayerStats 에 설정되어 있는
get : return 프로퍼티 변수를 참조해
체력을 증가시키거나 영혼 조각을 증가시키며

충돌한 아이템은 즉시 오브젝트 풀링 시스템에 있는
Queue 에 반납을 하고 비활성화 될 수 있도록
ObjectReturn 함수를 호출합니다.

문지기 Npc 와 상호작용을 하면 포탈을 열 수 있는데
입장 버튼에 OnClick 이벤트를 통해 인자를 받아
인자 값에 해당하는 포탈을 열 수 있도록 하였습니다.

```
void OnTriggerEnter(Collider other)
{
    if (other.tag == "DropItem")
    {
        ItemScript item = other.GetComponent<ItemScript>();
        switch (item.type)
        {
            case ItemScript.Type.Health:
                if (PlayerStats.Instance.Health >= PlayerStats.Instance.MaxHealth)
                {
                    return;
                }
                if (PlayerStats.Instance.Health < PlayerStats.Instance.MaxHealth)
                {
                    PlayerStats.Instance.Heal(1);
                    item.ObjectReturn();
                }
                break;
            case ItemScript.Type.Soul:
                SoundManager.Instance.SFXPlay("Soul acquire", AllGameManager.Instance.m_Clip[4]);
                PlayerStats.Instance.Addsoul();
                item.ObjectReturn();
                break;
        }
```

```
public void GateOpen(int GateNumber)
{
    for (int i = 0; i < 3; i++)
    {
        if (i == GateNumber)
        {
            m_Portal[i].SetActive(true);
            UIManager.Instance.SmithySystem(false, "Gate Keeper");
            AllGameManager.Instance.isWeaponShop = false;
        }
        else
            break;
    }
}
```

게임 시스템 관련

4-2. 대화 및 퀘스트 (1)

AllGameManger.cs 에서 플레이어의
대화 시스템을 관리합니다.

각 Npc 에 설정되어 있는 Npc Id 를 받아
TalkManager.cs 에 입력되어 있는
대화 텍스트들을 체크하고
텍스트의 Index 를 확인하여
순서대로 출력하게 하였습니다.

스토리 진행에 해당되지 않는
Npc 는 기본 대화가 나올 수 있도록 하기 위해
Npc 에 isNeedTalk 변수를 추가했습니다.

똑같은 Npc 지만 여러가지의 대화 텍스트들을
출력하도록 RandomtalkData 를 추가해
랜덤한 Index 의 Text 가 나오도록 하였습니다.

```
public void Action()
{
    ObjData objData = m_NearNpc.m_NearNpc.GetComponent<ObjData>();
    Talk(objData.id, objData.isNpc);

    if (objData.isNeedTalk)
    {

        UIManager.Instance.m_TalkPanel.SetActive(isTalkAction);
        UIManager.Instance.m_TalkImage.DOAnchorPosY(0, 0.75f).SetEase(Ease.OutQuad);
    }
}

void Talk(int id, bool isNpc)
{
    ObjData objData = m_NearNpc.m_NearNpc.GetComponent<ObjData>();
    int questTalkIndex = QuestManager.Instance.GetQuestTalkIndex(id);
    string talkData = TalkManager.Instance.GetTalk(id + questTalkIndex, m_TalkIndex);
    string RandomtalkData = TalkManager.Instance.GetTalk(id, Random.Range(0, 2));

    if (talkData == null)
    {
        QuestManager.Instance.m_QuestId = 10;

        if (id == 3000 && !objData.isNeedTalk)
            StationNpcInteraction();

        m_TalkIndex = 0;
        isTalkAction = false;
        objData.isNeedTalk = false;
        UIManager.Instance.m_TalkPanel.SetActive(isTalkAction);
        UIManager.Instance.m_TalkImage.DOAnchorPosY(-500, 0.75f).SetEase(Ease.OutQuad);
        return;
    }

    if (isNpc && objData.isNeedTalk)
    {
        UIManager.Instance.m_TalkText.text = talkData;
```

게임 시스템 관련

4-2. 대화 및 퀘스트 (2)

QuestManager.cs 에서
퀘스트 Id 와 List 를 관리합니다.

싱글톤을 사용하여 퀘스트 Id 를 참조할 수 있도록 구현하였습니다.

추가 , 검색 , 삭제 부분에 있어서
시간복잡도가 크지 않은
Dictionary 자료구조를 활용하였습니다.

자료형을 명확하게 설정해주는 점을 보고
헤시 테이블보다 속도면에서 좋은 성능을 보인다는 것을 느끼고
이와 같은 방법을 계속 사용하고 있습니다.

```
public int m_QuestId;

Dictionary<int, QuestData> m_QuestList;

private static QuestManager m_instance;
// 싱글톤
public static QuestManager Instance
{
    get
    {
        if (!m_instance)
        {
            m_instance = FindObjectOfType(typeof(QuestManager)) as QuestManager;

            if (m_instance == null)
                Debug.Log("No Singletone Obj");
        }
        return m_instance;
    }
}
```

```
void GenerateData()
{
    m_QuestList.Add(10, new QuestData("마을 사람들과 대화하기.", new int[] { 1000 , 2000 , 3000 , 4000
;
}

public int GetQuestTalkIndex(int id)
{
    return m_QuestId;
}
```

게임 시스템 관련

4-3. 비동기 로딩 시스템

LoadingSceneManager.cs 에서
비동기 로딩 시스템을 구현하였습니다.

일반적인 LoadScene 의 경우는
불러오는 도중에는 다른 작업을
할 수 없게 되지만

LoadSceneAsync 의 비동기 방식은
씬을 불러오는 도중에도
다른 작업이 가능하여 사용하게 되었습니다.

AsyncOperation 클래스의 객체인 op를 통해
로딩바를 구현하였고 로딩이 완료되고
설정해둔 시간이 지나면 씬을 불러오게 됩니다.

```
IEnumerator LoadSceneProcess()
{
    AsyncOperation op = SceneManager.LoadSceneAsync(nextScene); // 비동기 방식 > 다른작업 가능
    op.allowSceneActivation = false; // 씬을 넘길지 안넘길지

    float timer = -5f;
    while (!op.isDone)
    {
        yield return null;

        if (op.progress < 0.3f)
        {
            progressBar.fillAmount = op.progress;
        }
        else
        {
            timer += Time.unscaledDeltaTime;
            progressBar.fillAmount = Mathf.Lerp(0.3f, 1f, timer);
            if (progressBar.fillAmount >= 1f)
            {
                FadeInOutManager.Instance.OutStartFadeAnim(nextScene);
                op.allowSceneActivation = true;
                yield break;
            }
        }
    }
}
```

게임 시스템 관련

4-4. 오브젝트 풀링 (1)

오브젝트 풀링은 오브젝트를 Instantiate – Destroy 형식이 아닌
List 나 Queue 등에 저장해 사용하는 형식입니다.

ObjectsInfo 클래스에서 오브젝트들의 정보들을 관리하였습니다.
스크립트에 각 번호에 어느 Prefab 이 있는지 알 수 있도록 작성하였고

각각 등록된 오브젝트들의 생성 수가 다르기 때문에
모든 것을 m_ObjectPoolList 에 2차원 배열로 담아 두었습니다.

여러 곳에서 오브젝트들을 불러올 때 m_ObjectPoolList 를 사용합니다.

* 유튜브 포트폴리오 작업 이 후
진행된 작업입니다. 성능 개선을 위해
계속해서 수정 중에 있습니다.

```
[System.Serializable]
public class ObjectsInfo
{
    // Object 들의 정보들을 관리하는 Class
    public string m_ObjectName;
    public GameObject m_Prefab;
    public int m_Count;
}

public class ObjectPoolManager : MonoBehaviour
{
    public static ObjectPoolManager instance;


    /// <summary>
    /// 0 :: Slash Effect
    /// 1 :: Monster Hit Effect
    /// 2 :: Boss Red Bomb
    /// 3 :: Boss Blue Bomb
    /// 4 :: Boss Yellow Bomb
    /// 5 :: Boss White Bomb
    /// 6 :: Drop Soul
    /// 7 :: Drop Heal
    /// 8 :: Lantern Ball
    /// </summary>

    // Object List Save
    [SerializeField]
    ObjectsInfo[] m_ObjectInfos = null;

    [Header("Object Pool Location")]
    [SerializeField]
    Transform m_PoolParent;

    public List<Queue<GameObject>> m_ObjectPoolList;
```

게임 시스템 관련

4-4. 오브젝트 풀링 (2)

등록된 오브젝트들의 Count 만큼 Clone 을 생성해
m_ObjectPoolList 에 넘겨줍니다.

오브젝트의 Count 만큼 모두 비활성화를 한 다음
생성된 Clone 은 m_Queue 에 Enqueue 로 저장합니다.

오브젝트 풀링의 Queue 에서 Dequeue로
오브젝트를 꺼내어 사용한 후에는
DestroyObj 함수를 호출합니다.

사용한 오브젝트가 몇 번 Prefab 이였는지 넘겨주고
해당 오브젝트를 다시 Queue – Enqueue 로 저장하고
비활성화 시키는 방식 입니다.

```
void Start()
{
    m_ObjectPoolList = new List<Queue<GameObject>>();

    if(m_ObjectInfos != null)
        for (int i = 0; i < m_ObjectInfos.Length; i++)
        {
            m_ObjectPoolList.Add(InsertQueue(m_ObjectInfos[i]));
        }
}

Queue<GameObject> InsertQueue(ObjectsInfo prefab_objectInfo)
{
    Queue<GameObject> m_Queue = new Queue<GameObject>();

    for (int i = 0; i < prefab_objectInfo.m_Count; i++)
    {
        GameObject objectClone = Instantiate(prefab_objectInfo.m_Prefab) as GameObject;

        objectClone.SetActive(false);
        objectClone.transform.SetParent(m_PoolParent);
        m_Queue.Enqueue(objectClone);
    }

    return m_Queue;
}

public IEnumerator DestroyObj(float Seconds, int PoolNumber, GameObject Object)
{
    yield return new WaitForSeconds(Seconds);
    m_ObjectPoolList[PoolNumber].Enqueue(Object);
    Object.SetActive(false);
}
```

```
GameObject SlashEffect = ObjectPoolManager.instance.m_ObjectPoolList[0].Dequeue();
SlashEffect.SetActive(true);

SlashEffect.transform.position = m_SlashPos.position;
SlashEffect.transform.rotation = m_SlashPos.rotation;
Rigidbody bulletRigid = SlashEffect.GetComponent<Rigidbody>();
bulletRigid.velocity = m_SlashPos.forward * 20;

ObjectPoolManager.instance.StartCoroutine(ObjectPoolManager.instance.DestroyObj(1.5f, 0,
   SlashEffect));
```

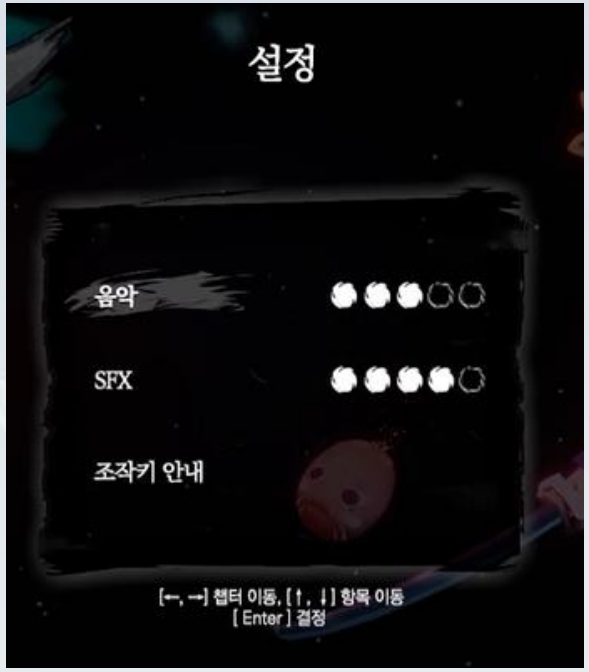
* ChoheeController.cs 에서
투사체를 생성하여 발사 할 때
사용한 ObjectPooling

유저 인터페이스 관련

5-1. 인터페이스 시스템

환경설정에서 방향키를 조절하여
사운드를 조절할 수 있습니다.

조절할 때 마다 체크를 하여
조절 값에 따라 게이지를 On / Off 합니다.



* 사운드 조절 창 - (환경 설정)

```
void SettingGageCtrl(int Gage)
{
    if(!m_SFX[0] && !m_BGM[0])
    {
        for (int i = Gage; i < 6; i++)
        {
            if (m_SettingMenuNumber == 1)
            {
                if(i == 0)
                    m_SFX[1].sprite = m_SettingGageOff;
                else if (i == Gage)
                    m_SFX[i].sprite = m_SettingGageOn;
                else if (i > Gage)
                    m_SFX[i].sprite = m_SettingGageOff;
            }

            else if (m_SettingMenuNumber == 2)
            {
                if (i == 0)
                    m_BGM[1].sprite = m_SettingGageOff;
                else if (i == Gage)
                    m_BGM[i].sprite = m_SettingGageOn;
                else if (i > Gage)
                    m_BGM[i].sprite = m_SettingGageOff;
            }
        }
    }
}
```

플레이어가 공격을 적중하면
스킬 (칼바람) 게이지가 충전됩니다.

PlayerStats 의 Slash 변수 값을 체크해
Slash 게이지와 동일한 값의 이미지와 (사용)
Slash 게이지보다 적은 값의 이미지와 (충전)
Slahs 게이지보다 높은 값의 이미지를 (미충전)
각각 다르게 두었습니다.

* 인게임 플레이어 UI

```
void SkillGageCtrl()
{
    int CurSlash = (int)PlayerStats.Instance.Slash;
    if (PlayerStats.Instance.Slash == CurSlash && !m_PlayerSkill[0])
    {
        for (int i = 1; i < 5; i++)
        {
            m_PlayerSkill[i].sprite = m_SkillGage[0];

            if(i == CurSlash)
                m_PlayerSkill[CurSlash].sprite = m_SkillGage[2];

            else if(i < CurSlash)
                m_PlayerSkill[i].sprite = m_SkillGage[1];
        }
    }
}
```

유저 인터페이스 관련

5-2. 사운드 시스템

SoundManager.cs 에서 사운드들을 관리합니다.

m_BgList[] 에 씬의 이름과 같은 사운드들을 넣고
씬이 교체될 때 마다 해당 씬의 이름을 가져와
동일한 이름의 Bgm 을 재생합니다.

싱글톤을 활용하여 구현하였고
SFXPlay 함수에 접근할 수 있도록 하여
필요에 따라 효과음을 재생시킬 수 있도록 하였습니다.

사운드의 음량 조절은 모두 Mixer 를 통해 조절됩니다.

```
private void OnSceneLoaded(Scene arg0, LoadSceneMode arg1)
{
    for (int i = 0; i < m_BgList.Length; i++)
    {
        Debug.Log(arg0.name);
        m_SceneName = arg0.name;
        if (arg0.name == m_BgList[i].name)
            BgSoundPlay(m_BgList[i]);
        else if (arg0.name == "LoadingScene")
            m_BgSound.Stop();
    }
}

public void BGSoundVolume(float val) // UI 볼륨 설정창
{
    m_Mixer.SetFloat("BGSoundVolume", val);
}

public void SFXSoundVolume(float val) // UI 볼륨 설정창
{
    m_Mixer.SetFloat("SFXVolume", val);
}

public void SFXPlay(string sfxName, AudioClip clip)
{
    GameObject go = new GameObject(sfxName + "Sound");
    AudioSource audiosource = go.AddComponent<AudioSource>();
    audiosource.outputAudioMixerGroup = m_Mixer.FindMatchingGroups("SFX")[0];
    audiosource.clip = clip;
    audiosource.Play();

    Destroy(go, clip.length);
}

public void BgSoundPlay(AudioClip clip)
{
    m_BgSound.outputAudioMixerGroup = m_Mixer.FindMatchingGroups("BGSound")[0];
    m_BgSound.clip = clip;
    m_BgSound.loop = true;
    m_BgSound.volume = 0.1f;
    m_BgSound.Play();
}
```

감사합니다.

길용현 – 아비규환 제작 코드 리뷰

https://youtu.be/SoN3IVONKro

https://github.com/Gilssom/Aura2_Test_Project.git

22_10_29 제작