경희대학교

최종 보고서

제　　목: Unity 엔진의 3D 게임디개발

전　　문: 컴퓨터 공학

이　　름: LIJIN

학　　번: 2021105555

지도교수: CHAONING ZHANG

Unity 엔진 기반 게임 디자인

요약: 시대가 발전함에 따라 다양한 3A 게임들이 계속해서 등장하고 있으며, 최근 인기를 끌고 있는 흑신화: 오공은 서유기를 소재로 한 훌륭한 액션 롤플레잉 게임입니다. Unity3D 는 그래픽, 오디오, 물리 충돌, 파티클 시스템, 메쉬 등 다양한 엔진 기능을 지원하는 게임 개발 엔진으로, 본 졸업 설계 프로젝트에 매우 적합합니다. 본 논문은 Unity2021.3.19f1 을 개발 환경으로 설정하고 C# 언어를 사용하여 싱글 플레이어 3D RPG 액션 어드벤처 게임인 <**협객**>을 개발 및 연구합니다. 본 프로젝트에서는 3D 액션 게임의 기본 요소인 캐릭터 이동, 점프, 공격 애니메이션 및 스크립트 등을 구현하며, Unity3D 를 활용한 게임 개발 방법과 기술을 단계적으로 적용합니다. 게임 제작 과정은 초기 기획, 자료 제작, 후반 설계, 코드 구현 등의 단계로 이루어지며, 최종적으로 하나의 완성된 3D 액션 게임을 개발하는 것을 목표로 합니다.

키워드: 게임 디자인；Unity3D；C# 언어；3D RPG 액션 어드벤처 게임

1. 서론

1.1 연구 배경

3D 액션 게임은 플레이어에게 긴장감과 스릴을 선사하며, 특히《흑신화: 오공》과 같은 작품에서는 게임이 단순한 오락을 넘어 도전과 몰입이 가득한 모험으로 다가옵니다. 이 게임의 스타일은 동양 신화의 깊이와 신비로움으로 가득하며, 사실적이고 웅장한 그래픽을 통해 문화적인 깊이가 느껴지는 세계를 만들어냅니다. 정교한 전투 시스템, 사실적인 캐릭터 표현, 점진적으로 상승하는 난이도 설정을 통해, 게임은 플레이어가 각 전투에서 최선을 다해 전투 기술을 연마하고 캐릭터의 성장 잠재력을 탐험하도록 자극합니다.

이러한 3D 게임 개발은 강력한 그래픽 렌더링 능력뿐 아니라 복잡한 물리 엔진, AI 상호작용 설계, 그리고 풍부한 스토리텔링과 장면 상호작용이 필요합니다. 디자이너는 이 과정에서 다양한 창의력을 발휘해, 복잡한 캐릭터 성장, 서사적인 스토리, 탐험 및 퍼즐 해결 등 다양한 RPG 요소를 녹여낼 수 있습니다. 이러한

게임은 제작진의 뛰어난 기술과 상상력을 보여줄 뿐 아니라, 플레이어가 마치 실제로 동양적이고 서사가 풍부한 이야기 속에 들어온 듯한 몰입감을 제공합니다.

1.2 연구 현황

1.2.1 중국 산업 현황

최근 중국의 3D RPG 액션 어드벤처 게임 시장은 빠르게 발전하며, 여러 고퀄리티 작품들이 등장하고 있습니다. 예를 들어, 오픈 월드 3D 액션 어드벤처 게임인《원신》은 국내외에서 큰 성공을 거두며 전 세계적인 주목을 받았습니다. 또 다른 예로는 사실적인 그래픽, 동양 신화 배경, 복잡한 전투 메커니즘으로 많은 관심을 받은 《흑신화: 오공》이 있습니다. 현재 국내의 많은 게임 개발사들은 동양 문화 요소를 탐구하고 이를 3D RPG 게임에 접목하여, 높은 수준의 그래픽과 혁신적인 게임 플레이로 플레이어들의 이목을 끌고자 노력하고 있습니다.

더불어, 국내 게임 시장의 지속적인 확장은 기술의 빠른 성장을 이끌고 있습니다. 많은 회사들이 자주 개발 엔진과 고급 그래픽 기술에 투자하여 게임의 시각적 표현과 상호 작용 경험을 향상시키고 있습니다. 과거와는 달리, 현재의 3D RPG 액션 어드벤처 게임은 스토리텔링, 캐릭터 성장, 풍부한 세계 탐험 요소를 더욱 중시합니다. 비록 시장 경쟁이 치열하지만, 국내의 3D RPG 액션 어드벤처 게임 산업은 계속해서 성장하며 광범위한 전망을 보여주고 있습니다.

1.2.2 해외 산업 현황

해외에서도 3D RPG 액션 어드벤처 게임의 연구와 개발은 큰 발전을 이루었습니다. 서구 시장에서는 《위쳐 3》, 《다크 소울》 시리즈, 《젤다의 전설: 야생의 숨결》과 같은 대표적인 3D RPG 게임들이 그래픽과 게임성에서 큰 성공을 거두었으며, 스토리와 몰입감 있는 경험 면에서도 많은 사랑을 받고 있습니다. 이러한 게임들은 보통 오픈 월드 또는 반오픈 월드 설정을 통해 흥미진진한 스토리와 자유로운 탐험 요소를 제공하여 플레이어에게 몰입감 있는 경험을 선사합니다.

해외 연구 분야에서는 3D RPG 액션 어드벤처 게임의 연구가 게임 메커니즘, 오픈 월드 디자인, AI 지능 및 플레이어 상호 작용에 집중되어 있습니다. 예를 들

어, 많은 게임 개발자들이 NPC 의 상호 작용 및 퀘스트 발동 메커니즘 최적화를 연구하여 플레이어의 몰입감을 높이고 있습니다. 동시에, 해외 게임 개발사들은 고급 그래픽 기술과 광원 효과를 통해 생동감 있는 게임 세계를 표현하는 데 주력하고 있습니다. 또한, 가상 현실(VR) 기술의 발전과 함께, 3D RPG 액션 어드벤처 게임의 몰입 경험에 대한 연구도 점점 깊어지고 있어, 앞으로 더 높은 상호 작용성을 갖춘 게임 경험을 제공할 것으로 기대됩니다.

종합적으로 볼 때, 해외의 3D RPG 액션 어드벤처 게임 산업은 기술 혁신과 깊이 있는 스토리 설계를 중시하며, 게임 경험의 최적화와 향상에 끊임없이 노력하여 전 세계 플레이어들이 사랑하는 인기 게임 장르 중 하나로 자리 잡고 있습니다.

2 개발 환경 및 기술

2.1 개발 환경

1. 운영 체제: Windows 10 64 비트

2. Unity3D 버전: 2021.3.19f1 버전

3. IDE: Visual Studio Code 1.94.2 버전

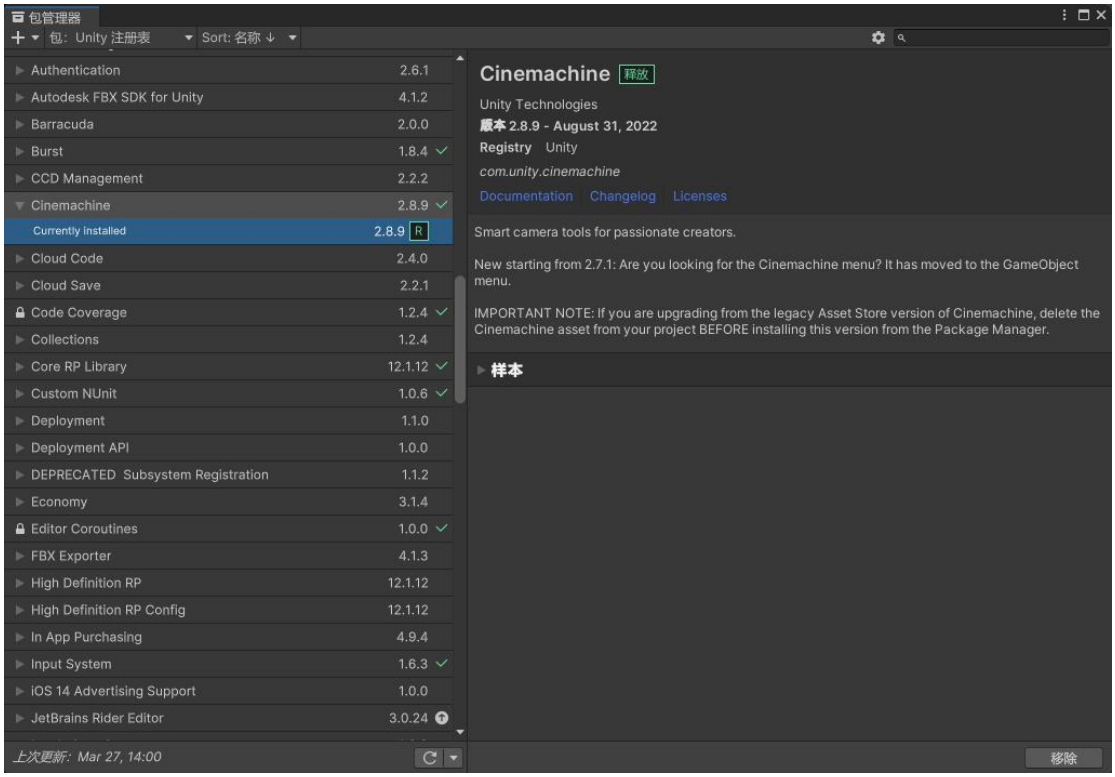
2.2 개발 언어

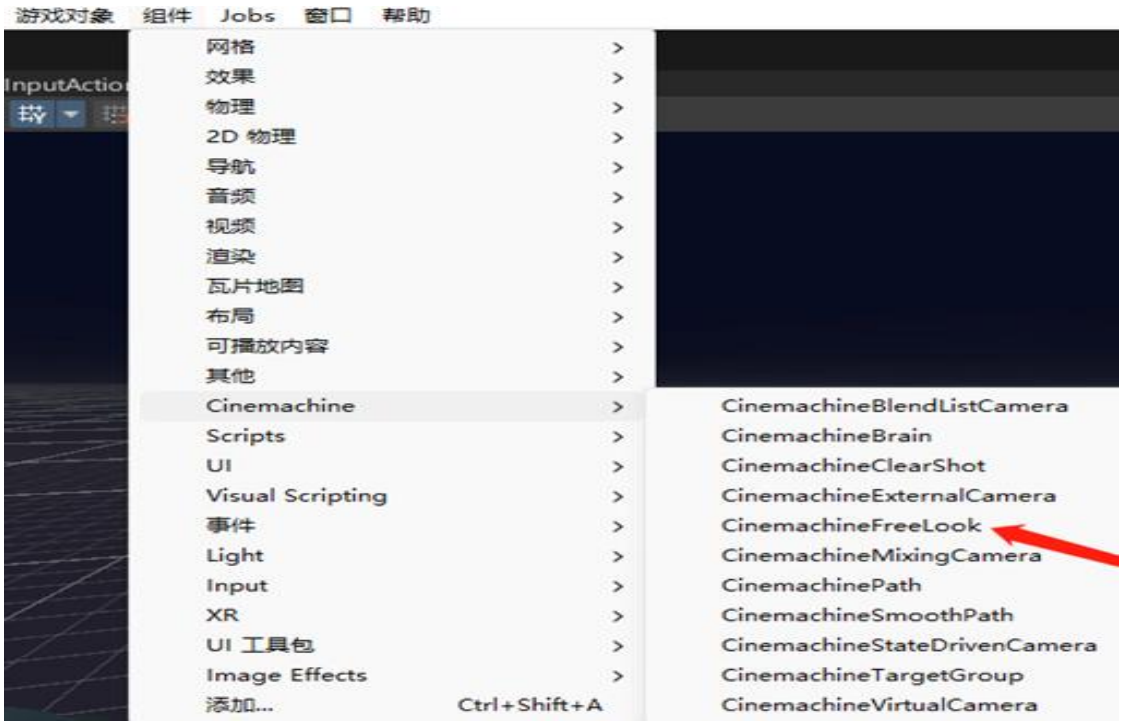
Unity3D 엔진은 C#과 JavaScript 를 지원합니다. 이번 졸업 프로젝트에서는 C# 프로그래밍 언어를 선택하였습니다. C#은 Windows, Mac, Linux, Android 및 iOS 에서 원활하게 실행되며, 호출이 편리하고 시스템성이 뛰어납니다.

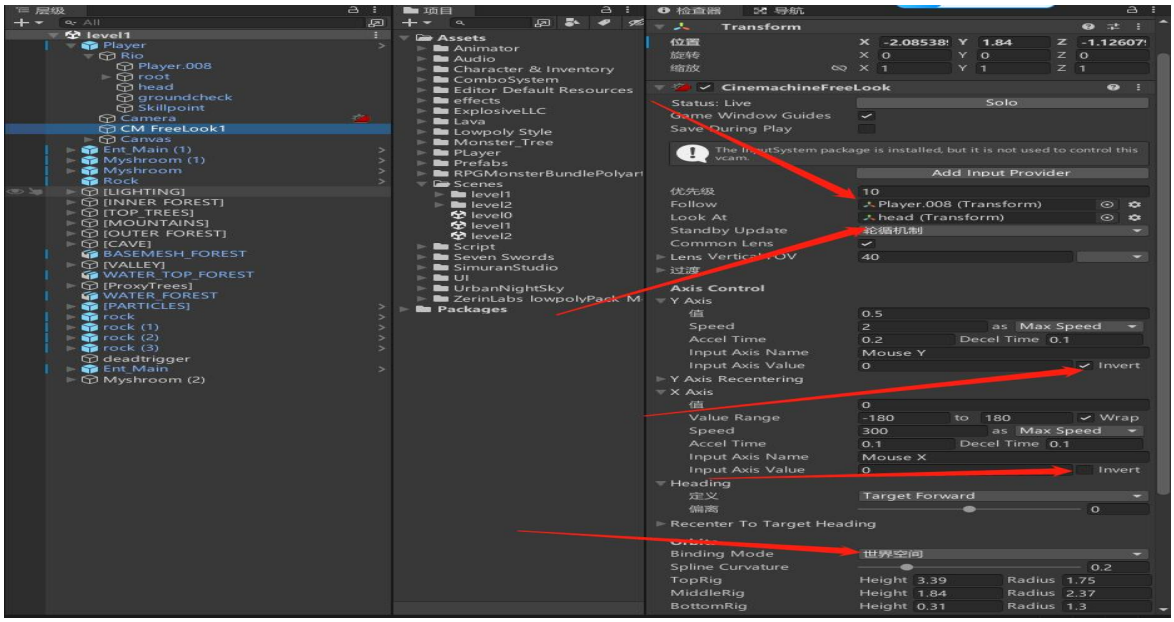
3.게임 기술 요점 및 구현 방식

3.1 캐릭터 이동 제어 부분

Cinemachine 플러그인을 사용하여 카메라 추적을 구현하고, CharacterController 컴포넌트를 통해 캐릭터 이동을 구현하였습니다. 플레이어의 입력에 따라 캐릭터가 목표 각도로 부드럽게 회전하며, 걷기 또는 정지 애니메이션이 재생됩니다.







Follow 에서 제공하는 것은 캐릭터 모델의 transform 이고, LookAt 에서 제공하는 것은 캐릭터 머리의 transform 입니다. 이렇게 하면 간단한 3 인칭 시점이 완성됩니다.

3.2 피격 부분

본 프로젝트에서는 주로 캐릭터 피격과 몬스터 피격을 다루며, 원리는 동일합니다. 여기서는 몬스터 피격에 대해 주로 설명합니다.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

6 个引用
public interface IAgent
{
    3 个引用
    public void GetDamage(float damge, Vector3 pos);
    3 个引用
    public void GetHit(int aniIndex,bool islightattack);
}
```

먼저, 인터페이스를 작성하고, 하나는 체력 감소를 처리하는 함수, 다른 하나는 애니메이션을 위한 함수를 정의합니다.

```
Unity 脚本|3 个引用
public class AgentHitBox : MonoBehaviour
{
    public GameObject Agent;
    IAgent target;
    Unity 消息|0 个引用
    private void Awake()
    {
        target = Agent.GetComponent<IAgent>();
    }
    2 个引用
    public void GetDamage(float damage,Vector3 pos)
    {
        target.GetDamage(damage, pos);
    }
    2 个引用
    public void GetHit(int aniIndex, bool islightattack)
    {
        target.GetHit(aniIndex,islightattack);
    }
}
```

그 다음으로, `Detection` 클래스를 작성하여 공격 감지가 필요한 오브젝트의 태그, 피격된 올바른 오브젝트 목록, 목록을 초기화하는 메서드, 그리고 오브젝트를 감지하는 메서드를 정의합니다.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity 脚本|3 个引用
public abstract class Detection : MonoBehaviour
{
    public string[] targetTags;
    public List<GameObject> wasHit = new List<GameObject>();
    1 个引用
    public void ClearWasHit() => wasHit.Clear();
    2 个引用
    public abstract List<Collider> GetDetection();

}
```

그 다음으로, CapsuleDetection 스크립트를 새로 생성합니다. 이 스크립트는 무기에 부착되어 실제 공격 감지를 담당합니다. 이 스크립트는 Detection 추상 클래스를 상속받습니다.

먼저, 추상 클래스의 getDetection 메서드를 재정의해야 합니다. 이 메서드는 먼저 감지된 충돌 객체를 저장하기 위한 빈 리스트 result 를 생성합니다. 그런 다음, Physics.OverlapCapsule 메서드를 사용하여 지정된 캡슐 모양과 교차하는 모든 콜라이더를 감지합니다.

감지된 각 콜라이더에 대해, 스크립트는 해당 콜라이더에 AgentHitBox 라는 컴포넌트가 포함되어 있는지 확인합니다. 또한, 이 컴포넌트와 연관된 에이전트 태그가 대상 태그 목록에 포함되어 있는지, 그리고 해당 에이전트가 이전에 감지된 적이 없는지도 확인합니다. 이러한 조건이 모두 충족되면, 스크립트는 해당 에이전트를 wasHit 목록에 추가하고, 이를 감지 결과로서 result 리스트에 추가합니다.

마지막으로, 이 메서드는 모든 감지된 충돌 객체를 포함한 result 리스트를 반환합니다.

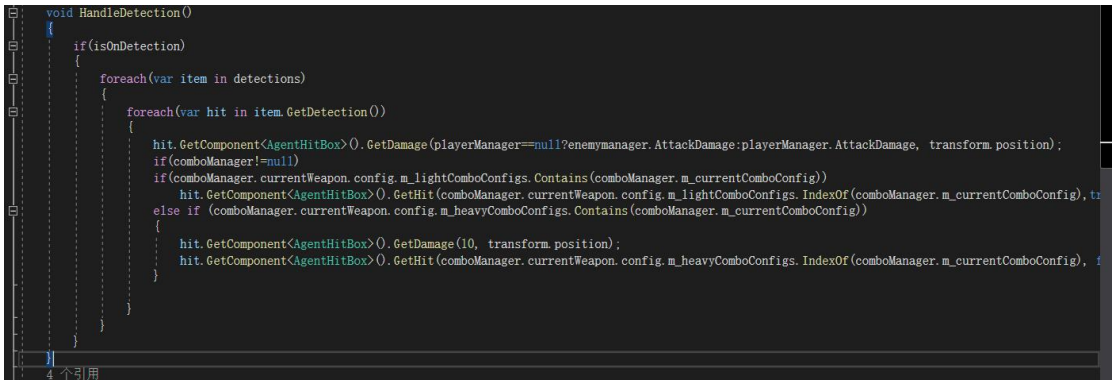
```
public override List<Collider> GetDetection()
{
    List<Collider> result = new List<Collider>();
    Collider[] hits = Physics.OverlapCapsule(startPoint.position, endPoint.position, radius);
    foreach(var item in hits)
    {
        AgentHitBox hitBox;
        if(item.TryGetComponent(out hitBox)&&hitBox.Agent&&targetTags.Contains(hitBox.Agent.tag)&&!wasHit.Contains(hitBox.Agent))
        {
            wasHit.Add(hitBox.Agent);
            result.Add(item);
        }
    }
    return result;
}
```

그 다음으로, 무기 활성화 및 비활성화를 감지하기 위한 weaponManager 스크립트를 작성합니다.

활성화 감지: 캐릭터가 감지 상태에 있는 경우, 감지된 모든 객체를 순회하며 해당 객체의 `GetDetection` 메서드가 반환한 충돌 리스트를 다시 순회합니다. 각 충돌 객체에 대해, `GetComponent` 메서드를 호출하여 해당 객체의 컴포넌트를 가져옵니다. 그런 다음, 해당 컴포넌트의 `GetDamage` 메서드를 호출하여 캐릭터의 현재 공격 데미지 값을 전달합니다.

만약 `comboManager` 컴포넌트가 존재하고, 현재 무기 설정에 경공격 조합이 포함되어 있다면, `GetHit` 메서드를 호출하여 경공격 조합의 인덱스와 경공격 여부를 나타내는 플래그를 전달합니다.

만약 현재 무기 설정에 중공격 조합이 포함되어 있다면, 충돌 객체에 추가적으로 10 의 데미지를 주고, `GetHit` 메서드를 호출하여 중공격 조합의 인덱스와 중공격 여부를 나타내는 플래그를 전달합니다.



비활성화 감지의 경우, 수집된 오브젝트 리스트를 바로 초기화하여 비웁니다.

```
public void ToggleDetection(bool value)
{
    isOnDetection = value;
    if (!value)
        foreach(var item in detections)
        {
            item.ClearWasHit();
        }
}
```

3.3 적 상태

적은 다음과 같은 상태를 가질 수 있습니다: 피격, 추격, 공격, 사망, 대기. 이러한 상태들 간에 조건 판단을 통해 상태를 전환해야 합니다. 여기서는 먼저 인터페이스를 작성해야 하며, 이 인터페이스의 함수들은 각각 다음을 수행합니다: 해당 상태에 진입 시 실행되는 함수, 해당 상태에서 update 함수에서 실행되는 함수, 해당 상태에서 fixedupdate 함수에서 실행되는 함수, 해당 상태에서 나올 때 실행되는 함수.

```
using System.Collections.Generic;
using UnityEngine;

8 个引用
public interface IState
{
    6 个引用
    void OnEnter();
    6 个引用
    void OnUpdate();
    6 个引用
    void OnFixedUpdate();
    6 个引用
    void OnExit();
}
```

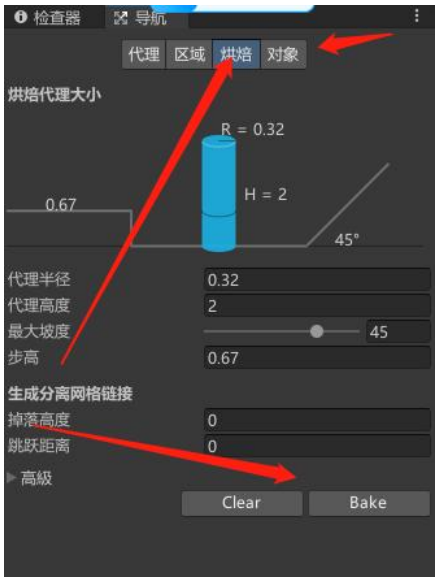
그 다음 enemyidlestate 스크립트를 작성하여 인터페이스를 상속하고, 대기 상태에서 실행되는 이벤트를 표현합니다.

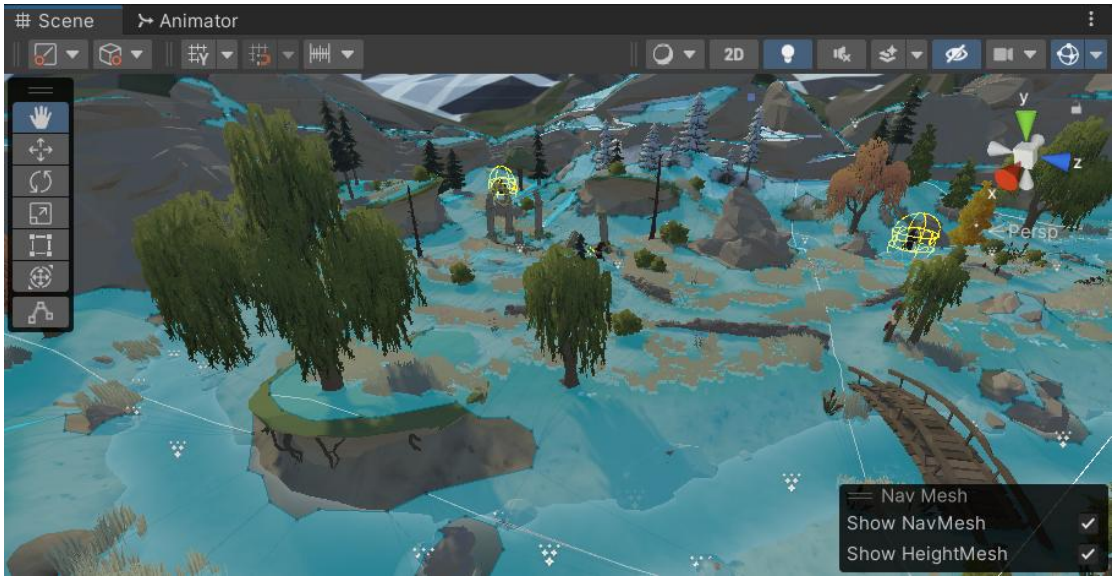
이 상태에 진입하면 대기 및 걷기 애니메이션이 재생되며, 적에게 속도를 부여합니다.

```
2 个引用
public void OnEnter()
{
    enemy.animator.CrossFadeInFixedTime("idle",1f);
    enemy.navMeshAgent.speed = enemy.speed;

}
```

본 프로젝트에서는 navmeshagent 컴포넌트를 사용하여 적의 이동을 구현하고 있습니다. 먼저 맵에 대한 베이킹이 필요하며, 파라미터를 조정하고 베이킹할 오브젝트를 선택한 후 바로 bake 를 진행하면 됩니다.





그 다음 적에게 몇 개의 지점을 경로로 설정하고, 해당 지점 근처에 도달하면 다음 지점으로 이동하도록 하여 계속 반복되도록 합니다.

enemymanager 스크립트에서

```
public void idlemove()
{
    if (Vector3.Distance(transform.position, pathPointList[currentPathPointIndex].position) > 0.5)
    {
        navMeshAgent.SetDestination(pathPointList[currentPathPointIndex].position);
        animator.Play("walk");
    }
    else
    {
        if (currentPathPointIndex < pathPointList.Count - 1)
            currentPathPointIndex++;
        else
            currentPathPointIndex = 0;
    }

}
```

대기 상태에서 플레이어와 적의 위치를 계속해서 확인합니다. 우선 플레이어의 위치 주변에 구형의 충돌 감지 범위를 정의하는데, 이 범위는 `chaseDistance` 변수로 지정됩니다. 이 범위는 `Physics.OverlapSphere` 메서드를 사용해 범위 내의 충돌체가 있는지 감지하며, 이 충돌체들은 플레이어 레이어(`playerlayer`)에 속해야 합니다. 만약 충돌체가 감지되면, 첫 번째 충돌체를 플레이어라고 가정하고, 그 변환기를 `player` 변수에 할당합니

다. 그 후 현재 객체와 플레이어 간의 거리를 계산해 distance 변수에 저장합니다. 만약 충돌체가 감지되지 않으면 player 변수는 null 로 설정됩니다.

```
public void GetPlayerTransform()
{
    Collider[] chaseClooiders = Physics.OverlapSphere(transform.position, chaseDistance,playerlayer);
    if(chaseClooiders.Length>0)
    {
        player = chaseClooiders[0].transform;
        distance=Vector3.Distance(player.position, transform.position) ;
    }
    else
    {
        player = null;
    }
}
1 个引用
```

다시 enemyidlestate로 돌아와서플레이어가 감지되면 플레이어와의 거리를 판단합니다. 공격 거리보다 멀면 추격 상태로 전환하고, 공격 거리보다 가까우면 공격 상태로 전환합니다. 또한 현재 체력이 0 보다 작으면 사망 상태로 전환합니다.

```
public void OnUpdate()
{
    enemy.GetPlayerTransform();
    enemy.idlemove();
    if (enemy.player!=null)
    {

        if (enemy.distance>enemy.attackDistance)
        {
            enemy.TransitionState(EnemyStateType.Chase);
        }
        else if(enemy.distance <=enemy.attackDistance)
        {
            enemy.TransitionState(EnemyStateType.Attack);
        }
    }
    if (enemy.CurrentHp <= 0)
        enemy.TransitionState(EnemyStateType.Death);
}
}
```

그 다음 enemychasestate 스크립트를 작성하여 인터페이스를 상속하고, 추격 상태에서 실행되는 이벤트를 표현합니다.

3.4 미니맵

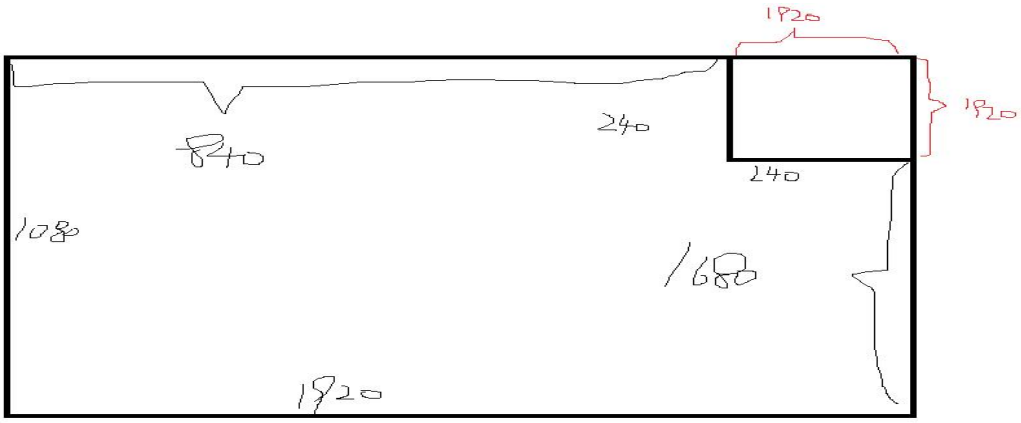
정사영 카메라를 생성하여 탑 뷰 시점으로 미니맵을 표시합니다. SetUIPos 메서드를 통해 적과 플레이어의 월드 좌표를 UI 좌표로 변환하여 미니맵에서 위치가 정확하게 표시되도록 합니다.

```
    for (int i = 0; i < enemyarray.Length; i++)
    {
        enemies.Add(enemyarray[i]);
        GameObject a = Instantiate(enemyUI, uiElement.parent.transform);
        enemyUIlist.Add(a);
    }
}
```

먼저 모든 enemy 태그의 객체를 배열에 추가한 후, 캔버스에 빨간 점을 생성하여 목록에 추가합니다.

```
2 个引用
public void SetUIPos(Transform traget, RectTransform UIrectTransform)
{
    Vector2 screenPos = cam.WorldToScreenPoint(traget.position);
    screenPos = new Vector2((screenPos.x * 0.125f + 1680), (screenPos.y  * 0.125f + 840));
    if (screenPos.x < 1680)
        screenPos.x = 1680;
    if (screenPos.y < 840)
        screenPos.y = 840;
    if(screenPos.y > 1080)
        screenPos.y = 1080;
    if (screenPos.x > 1920)
        screenPos.x = 1920;
    Vector2 localPos;
    RectTransformUtility.ScreenPointToLocalPointInRectangle(uiElement.parent.GetComponent<RectTransform>(), screenPos, null, out localPos);
    UIrectTransform.anchoredPosition = localPos;
}
```

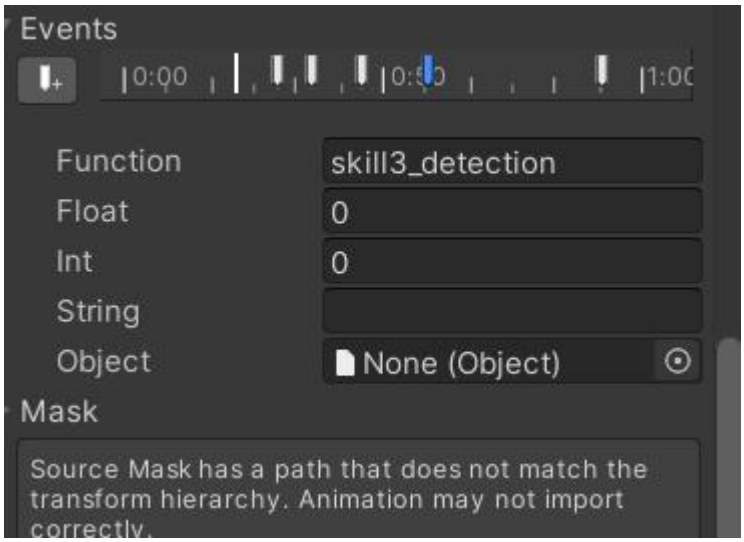
검은색은 화면 해상도이고, RawImage 의 크기는 240*240 이며, 렌더러 텍스처의 크기는 1920*1920 입니다. 예를 들어, 캐릭터가 직교 카메라의 중심 위치에 있을 때, 렌더러 텍스처의 크기가 1920*1920 이므로 (960*960) 좌표를 얻게 됩니다. 이 좌표를 화면에 배치하면 화면(1920*1080)에서 중간의 위쪽 위치에 놓이게 되므로, 위치를 계산할 필요가 있습니다.



검은색은 화면 해상도이고, RawImage 의 크기는 240*240 이며, 렌더러 텍스처의 크기는 1920*1920 입니다. 예를 들어, 캐릭터가 정교 카메라의 중심 위치에 있을 때, 렌더러 텍스처 크기가 1920*1920 이기 때문에 (960, 960) 좌표를 얻게 됩니다. 이 좌표를 화면에 배치하면 화면 해상도(1920*1080)에서는 중간 위쪽 위치에 놓이게 되므로, 위치를 계산해야 합니다.

3.5 스킬 발동

키 입력을 통해 스킬을 발동하며, 애니메이션 이벤트를 통해 특수 효과 인스턴스화, 범위 감지 및 스킬 상태 초기화를 수행하여 스킬 효과를 구현합니다. 범위 내 조건에 맞는 객체는 피해를 입게 됩니다.



범위 감지 코드는 PlayerManager 에 있습니다. 이 코드는 일정 범위 내의 충돌체를 수집한 후, 태그를 확인하여 조건에 맞는 경우 피해를 입히는 방식입니다.

```
0 个引用
public void skill2_detection()
{
    Collider[] hits = Physics.OverlapSphere(transform.position,  3);
    foreach (Collider c in hits)
    {
        if (c.gameObject.tag == "Enemy")
        {
            c.GetComponent<Enemymanager>().TransitionState(EnemyStateType.Hurt);
            c.GetComponent<Enemymanager>().CurrentHp -= 30f;
        }
    }
}
0 个引用
public void skill3_detection()
{
    Collider[] hits = Physics.OverlapCapsule(transform.position, skill1_endpos.position, 3);
    foreach (Collider c in hits)
    {
        if (c.gameObject.tag == "Enemy")
        {
            c.GetComponent<Enemymanager>().TransitionState(EnemyStateType.Hurt);
            c.GetComponent<Enemymanager>().CurrentHp -= 10f;
        }
    }
```

3.6 저장

JSON 파일을 사용하여 플레이어 데이터를 로컬에 저장하며, 데이터에는 레벨, 골드 등이 포함됩니다. 게임 시작 시 저장 파일을 읽어오며, 파일이 없을 경우 새 파일을 생성하여 플레이어의 진행 상황을 언제든지 복원할 수 있도록 합니다.

```
public void ReadData()
{
    string json;
    string filepath = Application.streamingAssetsPath + "/data.json";
    if (File.Exists(filepath))
    {
        using (StreamReader sr = new StreamReader(filepath))
        {
            json = sr.ReadToEnd();
            sr.Close();
        }
        peoples = JsonUtility.FromJson<personlist>(json);
        Debug.Log(peoples.people.Count);
    }
    else
        GenerateDataFile();
}
```

Person 클래스를 생성해야 합니다. 이 클래스에는 레벨, 골드, 약품 수량 등과 같은 캐릭터 정보가 포함됩니다.

```
public class person
{
    public string dataname;
    public int Level = 1;
    public int coin=0;
    public int HpMdc=0;
    public int AMdc=0;
    public float exp=0;
    public string time;
}
```

앞서 만든 Person 을 저장하기 위해 PersonList 클래스를 생성합니다.

```
public class personlist
{
    public List<person> people=new List<person>();
}
```

그 다음은 저장 파일을 생성하는 것으로, 새로운 Person 을 생성하여 앞의 목록에 추가합니다. 그리고 데이터를 JSON 파일에 저장하여 기록합니다.

4. 문제 정의

3D RPG 액션 어드벤처 게임을 개발하는 과정에서 몇 가지 주요 문제에 직면했습니다. 그 중에는 캐릭터 제어의 부드러움, 타격 피드백의 정확성, 적 AI 의 지능, 미니맵의 정확한 표시, 스킬의 실시간 반응, 저장 기능의 유효성이 포함됩니다. 이러한 문제들은 게임의 몰입도와 플레이 경험에 직접적인 영향을 미칩니다.

5. 해결 방법

1. 캐릭터 이동 제어: Cinemachine 플러그인을 사용하여 카메라 추적을 구현하고, CharacterController 컴포넌트로 캐릭터 이동을 처리하였습니다. SmoothDampAngle 을 사용하여 회전의 부드러움을 확보하여 캐릭터 제어의 경험을 향상시켰습니다.

2. 타격 피드백: 인터페이스를 통해 체력 감소 및 애니메이션 피드백 메커니즘을 설계하고, CapsuleDetection 을 통해 공격 범위 내의 충돌 객체를 감지하여 타격 피드백의 실시간성과 정확성을 보장하였습니다.

3. 적 AI: 다중 상태 AI 시스템(대기, 추적, 공격 등)을 설계하고, NavMeshAgent 컴포넌트를 통해 경로 계획을 구현하여 적의 지능을 높였습니다.

4. 미니맵 표시: 정사영 카메라와 SetUIPos 메서드를 통해 플레이어와 적의 월드 좌표를 정확히 UI 좌표로 변환하여 미니맵이 게임 상황을 정밀하게 반영할 수 있게 하였습니다.

5. 스킬 발동: 키 입력으로 스킬을 발동하고 애니메이션 이벤트를 통해 특수 효과를 인스턴스화, 범위 감지 및 스킬 상태 초기화를 수행하여 스킬 효과가 실시간으로 부드럽게 작동하도록 구현하였습니다.

6. 저장 기능: JSON 파일을 통해 플레이어 데이터를 로컬에 저장하여, 게임 시작 시 저장된 진행 상황을 복원할 수 있게 하여 지속적인 게임 경험을 제공합니다.

6. 결론과 미래 계획

결론

본 프로젝트는 Unity3D 엔진을 사용하여 3D RPG 액션 어드벤처 게임을 구축하였으며, 부드러운 캐릭터 제어 시스템, 정확한 타격 피드백, 지능적인 적 AI, 효율적인 미니맵과 저장 시스템을 성공적으로 구현하였습니다. 각 모듈이 모듈화 설계를 통해 독립적으로 작동하여 전체 구조가 명확하고 기능이 완전하게 유지되도록 하였으며, 이를 통해 플레이어에게 우수한 게임 경험을 제공하였습니다.

미래 계획

이번 졸업 프로젝트를 통해 Unity3D 엔진 사용, 캐릭터 제어 시스템 설계, AI 개발, 미니맵 및 저장 시스템 구현 등 3D 게임 개발의 핵심 기술을 더욱 숙달할 수 있었습니다. 이러한 실무 경험을 통해 게임 개발 프로세스에 대한 깊은 이해와 함께 프로그래밍 및 문제 해결 능력을 향상시켰으며, 향후 회사에 입사하여 업무에 기여할 수 있는 견고한 기반을 다졌습니다.