Unity Certified Associate 자격증 취득반 9회차 - 코드

김 영 득

2023. 6. 29

GameManager 객체 생성

GameManager 스크립트를 다음과 같이 작성한다

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

public class GameManager : MonoBehaviour
{
    // 몬스터가 출현할 위치를 저장할 배열
    public Transform[] points;


    // Start is called before the first frame update
    void Start()
    {
        // SpawnPointGroup 게임오브젝트의 Transform 컴포넌트 추출
        Transform spawnPointGroup = GameObject.Find("SpawnPointGroup")?.transform;

        // SpawnPointGroup 하위에 있는 모든 차일드 게임오브젝트의 Transform 컴포넌트 추출
        points = spawnPointGroup?.GetComponentsInChildren<Transform>();
    }

}
```

작성을 완료한 후 스크립트를 하이러키 뷰의 GameMgr에 추가하고 유니티를 실행한다 다음과 같이 points배열에 SpawnPointGroup 하위에 있는 모든 게임오브젝트의 Transform 컴포넌트가 저장된 것을 확인할 수 있다

GameManager 객체 생성

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

public class GameManager : MonoBehaviour
{
    // 몬스터가 출현할 위치를 저장할 List 타입 변수
    public List<Transform> points = new List<Transform>();


    // Start is called before the first frame update
    void Start()
    {
        // SpawnPointGroup 게임오브젝트의 Transform 컴포넌트 추출
        Transform spawnPointGroup = GameObject.Find("SpawnPointGroup")?.transform;

        // SpawnPointGroup 하위에 있는 모든 차일드 게임오브젝트의 Transform 컴포넌트 추출
        spawnPointGroup?.GetComponentsInChildren<Transform>(points);
    }

}
```

변수 선언부의 배열을 List 타입의 변수로 수정한다
List 타입은 저장하려는 데이터의 형식을 형식 매개변수인 <T>에 지정해야 한다
따라서 List<Transform>은 Transfomr타입의 데이터를 저장한다는 의미다

Invoke, InvokeRepeate 함수

일정 시간 간격으로 몬스터를 불규칙한 위치에 생성하는 로직을 구현한다
GameManager 스크립트를 다음과 같이 작성한다

GameManager 스크립트의 선언부에 Monster 프리팹을 연결할 변수와 생성 간격을 저장할 변수를 선언했다

```
// 몬스터 프리팹을 연결할 변수
public GameObject monster;

// 몬스터의 생성 간격
public float createTime = 3.0f;
```

주인공 캐릭터가 사망하거나 게임이 종료되는 조건을 만족하면 몬스터를 생성하는 로직을 정지시켜야 한다
정지시키는 로직의 함수를 만들어 호출하는 방식도 있지만, C#에서 제공하는 프로퍼티 문법을 사용해보자
프로퍼티는 객체지향 언어의 특징인 데이터 은닉성을 유지하면서 해당 데이터를 안전하게 외부에 노출하는 방법이다

Private 접근 제한자로 선언한 isGameOver 변수는 외부에 노출되지 않는다
즉, 다른 클래스에서 이 변수를 직접 읽고 쓸 수 없다
대신 IsGameOver라는 프로퍼티를 선언해 isGameOver변수를 간접적으로 다른 클래스에 노출한다

```
public class GameManager : MonoBehaviour
{
    // 몬스터가 출현할 위치를 저장할 List 타입 변수
    public List<Transform> points = new List<Transform>();

    // 몬스터 프리팹을 연결할 변수
    public GameObject monster;

    // 몬스터의 생성 간격
    public float createTime = 3.0f;

    // 게임의 종료 여부를 저장할 멤버 변수
    private bool isGameOver;

    // 게임의 종료 여부를 저장할 프로퍼티
    public bool IsGameOver
    {
        get { return isGameOver; }
        set
        {
            isGameOver = value;
            if (isGameOver)
            {
                CancelInvoke("CreateMonster");
            }
        }
    }

    // Start is called before the first frame update
    void Start()
    {
        // SpawnPointGroup 게임오브젝트의 Transform 컴포넌트 추출
        Transform spawnPointGroup = GameObject.Find("SpawnPointGroup")?.transform;

        // SpawnPointGroup 하위에 있는 모든 차일드 게임오브젝트의 Transform 컴포넌트 추출
        foreach (Transform point in spawnPointGroup)
        {
            points.Add(point);
        }

        // 일정한 시간 간격으로 함수를 호출
        InvokeRepeating("CreateMonster", 2.0f, createTime);
    }

    void CreateMonster()
    {
        // 몬스터의 불규칙한 생성 위치 산출
        int idx = Random.Range(0, points.Count);

        // 몬스터 프리팹 생성
        Instantiate(monster, points[idx].position, points[idx].rotation);
    }
}
```

싱글턴 디자인 패턴

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

public class GameManager : MonoBehaviour
{
    // 몬스터가 출현할 위치를 저장할 List 타입 변수
    public List<Transform> points = new List<Transform>();

    // 몬스터 프리팹을 연결할 변수
    public GameObject monster;

    // 몬스터의 생성 간격
    public float createTime = 3.0f;

    // 게임의 종료 여부를 저장할 멤버 변수
    private bool isGameOver;

    // 게임의 종료 여부를 저장할 프로퍼티
    public bool IsGameOver
    {
        get { return isGameOver; }
        set
        {
            isGameOver = value;
            if (isGameOver)
            {
                CancelInvoke("CreateMonster");
            }
        }
    }

    // 싱글턴 인스턴스 선언
    public static GameManager instance = null;

    // 스크립트가 실행되면 가장 먼저 호출되는 유니티 이벤트 함수
    void Awake()
    {
        // instance가 할당되지 않았을 경우
        if (instance == null)
        {
            instance = this;
        }
        // instance에 할당된 클래스의 인스턴스가 다를 경우 새로 생성된 클래스를 의미함
        else if (instance != this)
        {
            Destroy(this.gameObject);
        }
        // 다른 씬으로 넘어가더라도 삭제하지 않고 유지함
        DontDestroyOnLoad(this.gameObject);
    }

    // Start is called before the first frame update
    void Start()
    {
        // SpawnPointGroup 게임오브젝트의 Transform 컴포넌트 추출
        Transform spawnPointGroup = GameObject.Find("SpawnPointGroup")?.transform;

        // SpawnPointGroup 하위에 있는 모든 차일드 게임오브젝트의 Transform 컴포넌트 추출
        foreach (Transform point in spawnPointGroup)
        {
            points.Add(point);
        }

        // 일정한 시간 간격으로 함수를 호출
        InvokeRepeating("CreateMonster", 2.0f, createTime);
    }

    void CreateMonster()
    {
        // 몬스터의 불규칙한 생성 위치 산출
        int idx = Random.Range(0, points.Count);

        // 몬스터 프리팹 생성
        Instantiate(monster, points[idx].position, points[idx].rotation);
    }
}
```

오브젝트 풀링

다음과 같이 GameManager.cs스크립트를 수정한다

```
public class GameManager : MonoBehaviour
{
    // 몬스터가 출현할 위치를 저장할 List 타입 변수
    public List<Transform> points = new List<Transform>();

    // 몬스터를 미리 생성해 저장할 리스트 자료형
    public List<GameObject> monsterPool = new List<GameObject>();

    // 오브젝트 풀(Object Pool)에 생성할 몬스터의 최대 개수
    public int maxMonsters = 10;

    // 몬스터 프리팹을 연결할 변수
    public GameObject monster;

    // 몬스터의 생성 간격
    public float createTime = 3.0f;

    // 게임의 종료 여부를 저장할 멤버 변수
    private bool isGameOver;

    // 게임의 종료 여부를 저장할 프로퍼티
    public bool IsGameOver
    {
        get { return isGameOver; }
        set
        {
            isGameOver = value;
            if (isGameOver)
            {
                CancelInvoke("CreateMonster");
            }
        }
    }

    // 싱글턴 인스턴스 선언
    public static GameManager instance = null;

    // 스크립트가 실행되면 가장 먼저 호출되는 유니티 이벤트 함수
    void Awake()
    {
        // instance가 할당되지 않았을 경우
        if (instance == null)
        {
            instance = this;
        }
        // instance에 할당된 클래스의 인스턴스가 다를 경우 새로 생성된 클래스를 의미함
        else if (instance != this)
        {
            Destroy(this.gameObject);
        }
        // 다른 씬으로 넘어가더라도 삭제하지 않고 유지함
        DontDestroyOnLoad(this.gameObject);
    }

    // Start is called before the first frame update
    void Start()
    {
        // 몬스터 오브젝트 풀 생성
        CreateMonsterPool();

        // SpawnPointGroup 게임오브젝트의 Transform 컴포넌트 추출
        Transform spawnPointGroup = GameObject.Find("SpawnPointGroup")?.transform;

        // SpawnPointGroup 하위에 있는 모든 차일드 게임오브젝트의 Transform 컴포넌트 추출
        foreach (Transform point in spawnPointGroup)
        {
            points.Add(point);
        }

        // 일정한 시간 간격으로 함수를 호출
        InvokeRepeating("CreateMonster", 2.0f, createTime);
    }

    void CreateMonster()
    {
        // 몬스터의 불규칙한 생성 위치 산출
        int idx = Random.Range(0, points.Count);

        // 몬스터 프리팹 생성
        Instantiate(monster, points[idx].position, points[idx].rotation);
    }

    // 오브젝트 풀에 몬스터 생성
    void CreateMonsterPool()
    {
        for (int i = 0; i < maxMonsters; i++)
        {
            // 몬스터 생성
            var _monster = Instantiate<GameObject>(monster);
            // 몬스터의 이름을 지정
            _monster.name = $"Monster_{i:00}";
            // 몬스터 비활성화
            _monster.SetActive(false);

            // 생성한 몬스터를 오브젝트 풀에 추가
            monsterPool.Add(_monster);
        }
    }

}
```

오브젝트 풀링

오브젝트 풀의 초기화 및 생성이 완료됐다
이제 비활성화된 상태로 생성된 몬스터를 하나씩 꺼내서 사용하는 로직을 완성해보자
GameManager 스크립트를 다음과 같이 수정한다

```
public class GameManager : MonoBehaviour
{
    // 몬스터가 출현할 위치를 저장할 List 타입 변수
    public List<Transform> points = new List<Transform>();

    // 몬스터를 미리 생성해 저장할 리스트 자료형
    public List<GameObject> monsterPool = new List<GameObject>();

    // 오브젝트 풀(Object Pool)에 생성할 몬스터의 최대 개수
    public int maxMonsters = 10;

    // 몬스터 프리팹을 연결할 변수
    public GameObject monster;

    // 몬스터의 생성 간격
    public float createTime = 3.0f;

    // 게임의 종료 여부를 저장할 멤버 변수
    private bool isGameOver;

    // 게임의 종료 여부를 저장할 프로퍼티
    public bool IsGameOver
    {
        get { return isGameOver; }
        set
        {
            isGameOver = value;
            if (isGameOver)
            {
                CancelInvoke("CreateMonster");
            }
        }
    }

    // 싱글턴 인스턴스 선언
    public static GameManager instance = null;

    // 스크립트가 실행되면 가장 먼저 호출되는 유니티 이벤트 함수
    void Awake()
    {
        // instance가 할당되지 않았을 경우
        if (instance == null)
        {
            instance = this;
        }
        // instance에 할당된 클래스의 인스턴스가 다를 경우 새로 생성된 클래스를 의미함
        else if (instance != this)
        {
            Destroy(this.gameObject);
        }
        // 다른 씬으로 넘어가더라도 삭제하지 않고 유지함
        DontDestroyOnLoad(this.gameObject);
    }

    // Start is called before the first frame update
    void Start()
    {
        // 몬스터 오브젝트 풀 생성
        CreateMonsterPool();

        // SpawnPointGroup 게임오브젝트의 Transform 컴포넌트 추출
        Transform spawnPointGroup =
GameObject.Find("SpawnPointGroup")?.transform;

        // SpawnPointGroup 하위에 있는 모든 차일드 게임오브젝트의
Transform 컴포넌트 추출
        foreach (Transform point in spawnPointGroup)
        {
            points.Add(point);
        }

        // 일정한 시간 간격으로 함수를 호출
        InvokeRepeating("CreateMonster", 2.0f, createTime);
    }

    void CreateMonster()
    {
        // 몬스터의 불규칙한 생성 위치 산출
        int idx = Random.Range(0, points.Count);

        // 몬스터 프리팹 생성
        //Instantiate(monster, points[idx].position,
points[idx].rotation);

        // 오브젝트 풀에서 몬스터 추출
        GameObject _monster = GetMonsterInPool();
        // 추출한 몬스터의 위치와 회전을 설정
        _monster?.transform.SetPositionAndRotation(points[idx].position,

points[idx].rotation);

        // 추출한 몬스터를 활성화
        _monster?.SetActive(true);
    }

    // 오브젝트 풀에 몬스터 생성
    void CreateMonsterPool()
    {
        for (int i = 0; i < maxMonsters; i++)
        {
            // 몬스터 생성
            var _monster = Instantiate<GameObject>(monster);

            // 몬스터의 이름을 지정
            _monster.name = $"Monster_{i:00}";

            // 몬스터 비활성화
            _monster.SetActive(false);

            // 생성한 몬스터를 오브젝트 풀에 추가
            monsterPool.Add(_monster);
        }
    }

    // 오브젝트 풀에서 사용 가능한 몬스터를 추출해 반환하는 함수
    public GameObject GetMonsterInPool()
    {
        // 오브젝트 풀의 처음부터 끝까지 순회
        foreach (var _monster in monsterPool)
        {
            // 비활성화 여부로 사용 가능한 몬스터를 판단
            if (_monster.activeSelf == false)
            {
                // 몬스터 반환
                return _monster;
            }
        }
        return null;
    }
}
```

오브젝트 풀링

1. 각종 컴포넌트를 할당하는 로직을 맨 먼저 수행하기 위해 Start 함수명을 Awake로 변경한다
2. 기존 Start 함수에 있던 코루틴 실행 코드는 OnEbable 함수로 옮긴다
3. MonsterAction 함수에서 일정 시간이 지난 몬스터를 비활성화한다

다음과 같이 MonsterCtrl 스크립트의 Start 함수를 Awake로 변경한다

```
void Awake()
    {
        // 몬스터의 Transform 할당
        monsterTr = GetComponent<Transform>();

        // 추적 대상인 Player의 Transform 할당
        playerTr = GameObject.FindWithTag("PLAYER").GetComponent<Transform>();

        // NavMeshAgent 컴포넌트 할당
        agent = GetComponent<NavMeshAgent>();

        // Animator 컴포넌트 할당
        anim = GetComponent<Animator>();

        // BloodSprayEffect 프리팹 로드
        bloodEffect = Resources.Load<GameObject>("BloodSprayEffect");
    }
```

오브젝트 풀링

Start 함수를 Awake 함수로 변경한 후 2개의 StarCoroutine 함수를 OnEnable 함수로 옮긴다

```
// 스크립트가 활성화될 때마다 호출되는 함수
void OnEnable()
{
    // 이벤트 발생 시 수행할 함수 연결
    PlayerCtrl.OnPlayerDie += this.OnPlayerDie;

    // 몬스터의 상태를 체크하는 코루틴 함수 호출
    StartCoroutine(CheckMonsterState());
    // 상태에 따라 몬스터의 행동을 수행하는 코루틴 함수 호출
    StartCoroutine(MonsterAction());
}
```

OnEnable 함수는 스크립트 또는 게임오브젝트가 비활성화된 상태에서 다시 활성화될 때마다 발생하는 유니티 콜백 함수다
따라서 코루틴 함수를 실행하는 부분을 OnEnable 함수로 옮겨 오브젝트 풀에서 재사용 하기 위해 활성화될 때 CheckMonsterState와 MonsterAction 코루틴 함수가 다시 호출되게 한다
또한, Start 함수를 Awake 함수로 변경한 이유는 OnEnable 함수가 Start 함수보다 먼저 수행되어 각종 컴포넌트가 연결되기 이전에 CheckMonsterState와 MonsterAction코루틴 함수가 수행될 경우 연결되지 않는 컴포넌트를 참조하는 오류가 발생하기 때문이다

오브젝트 풀링

MonsterAction 함수를 다음과 같이 수정한다
Switch – case 구문의 Stage.DIE 부분을 수정한다

```
// 몬스터의 상태에 따라 몬스터의 동작을 수행
    IEnumerator MonsterAction()
    {
        while (!isDie)
        {
            switch (state)
            {
                // IDLE 상태
                case State.IDLE:
                    // 추적 중지
                    agent.isStopped = true;

                    // Animator의 IsTrace 변수를 false로 설정
                    anim.SetBool(hashTrace, false);

                    break;

                // 추적 상태
                case State.TRACE:
                    // 추적 대상의 좌표로 이동 시작
                    agent.SetDestination(playerTr.position);

                    agent.isStopped = false;

                    // Animator의 IsTrace 변수를 true로 설정
                    anim.SetBool(hashTrace, true);

                    // Animator의 IsAttack 변수를 false로 설정
                    anim.SetBool(hashAttack, false);

                    break;

                // 공격 상태
                case State.ATTACK:
                    // Animator의 IsAttack 변수를 true로 설정
                    anim.SetBool(hashAttack, true);
                    break;

                // 사망
                case State.DIE:
                    isDie = true;

                    // 추적 정지
                    agent.isStopped = true;

                    // 사망 애니메이션 실행
                    anim.SetTrigger(hashDie);

                    // 몬스터의 Collider 컴포넌트 비활성화
                    GetComponent<CapsuleCollider>().enabled = false;

                    // 일정 시간 대기 후 오브젝트 풀링으로 환원
                    yield return new WaitForSeconds(3.0f);

                    // 사망 후 다시 사용할 때를 위해 hp 값 초기화
                    hp = 100;
                    isDie = false;

                    // 몬스터의 Collider 컴포넌트 활성화
                    GetComponent<CapsuleCollider>().enabled = true;
                    // 몬스터를 비활성화
                    this.gameObject.SetActive(false);

                    break;
            }
            yield return new WaitForSeconds(0.3f);
        }
    }
```

오브젝트 풀링

다음 MonsterCtrl 스크립트는 최종 완성된 전체 코드다

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
// 내비게이션 기능을 사용하기 위해 추가해야 하는 네임스페이스
using UnityEngine.AI;

public class MonsterCtrl : MonoBehaviour
{
    // 몬스터의 상태 정보
    public enum State
    {
        IDLE,
        TRACE,
        ATTACK,
        DIE
    }

    // 몬스터의 현재 상태
    public State state = State.IDLE;
    // 추적 사정거리
    public float traceDist = 10.0f;
    // 공격 사정거리
    public float attackDist = 2.0f;
    // 몬스터의 사망 여부
    public bool isDie = false;

    // 컴포넌트의 캐시를 처리할 변수
    private Transform monsterTr;
    private Transform playerTr;
    private NavMeshAgent agent;
    private Animator anim;

    // Animator 파라미터의 해시값 추출
    private readonly int hashTrace = Animator.StringToHash("IsTrace");
    private readonly int hashAttack = Animator.StringToHash("IsAttack");
    private readonly int hashHit = Animator.StringToHash("Hit");
    private readonly int hashPlayerDie = Animator.StringToHash("PlayerDie");
    private readonly int hashSpeed = Animator.StringToHash("Speed");
    private readonly int hashDie = Animator.StringToHash("Die");

    // 혈흔 효과 프리팹
    private GameObject bloodEffect;


    // 몬스터 생명 변수
    private int hp = 100;
    // 스크립트가 활성화될 때마다 호출되는 함수
    void OnEnable()
    {
        // 이벤트 발생 시 수행할 함수 연결
        PlayerCtrl.OnPlayerDie += this.OnPlayerDie;

        // 몬스터의 상태를 체크하는 코루틴 함수 호출
        StartCoroutine(CheckMonsterState());
        // 상태에 따라 몬스터의 행동을 수행하는 코루틴 함수 호출
        StartCoroutine(MonsterAction());
    }

// 스크립트가 비활성화될 때마다 호출되는 함수
    void OnDisable()
    {
        // 기존에 연결된 함수 해제
        PlayerCtrl.OnPlayerDie -= this.OnPlayerDie;
    }

    void Awake()
    {
        // 몬스터의 Transform 할당
        monsterTr = GetComponent<Transform>();

        // 추적 대상인 Player의 Transform 할당
        playerTr = GameObject.FindWithTag("PLAYER").GetComponent<Transform>();

        // NavMeshAgent 컴포넌트 할당
        agent = GetComponent<NavMeshAgent>();

        // Animator 컴포넌트 할당
        anim = GetComponent<Animator>();

        // BloodSprayEffect 프리팹 로드
        bloodEffect = Resources.Load<GameObject>("BloodSprayEffect");
    }



    // 일정한 간격으로 몬스터의 행동 상태를 체크
    IEnumerator CheckMonsterState()
    {
        while (!isDie)
        {
            // 0.3초 동안 중지(대기)하는 동안 제어권을 메시지 루프에 양보
            yield return new WaitForSeconds(0.3f);

            // 몬스터의 상태가 DIE일 때 코루틴을 종료
            if (state == State.DIE) yield break;

            // 몬스터와 주인공 캐릭터 사이의 거리 측정
            float distance = Vector3.Distance(playerTr.position, monsterTr.position);

            // 공격 사정거리 범위로 들어왔는지 확인
            if (distance <= attackDist)
            {
                state = State.ATTACK;
            }
            // 추적 사정거리 범위로 들어왔는지 확인
            else if (distance <= traceDist)
            {
                state = State.TRACE;
            }
            else
            {
                state = State.IDLE;
            }
        }
    }


    // 몬스터의 상태에 따라 몬스터의 동작을 수행
    IEnumerator MonsterAction()
    {
        while (!isDie)
        {
            switch (state)
            {
                // IDLE 상태
                case State.IDLE:
                    // 추적 중지
                    agent.isStopped = true;
                    // Animator의 IsTrace 변수를 false로 설정
                    anim.SetBool(hashTrace, false);
                    break;
                // 추적 상태
                case State.TRACE:
                    // 추적 대상의 좌표로 이동 시작
                    agent.SetDestination(playerTr.position);
                    agent.isStopped = false;
                    // Animator의 IsTrace 변수를 true로 설정
                    anim.SetBool(hashTrace, true);
                    // Animator의 IsAttack 변수를 false로 설정
                    anim.SetBool(hashAttack, false);
                    break;

                // 공격 상태
                case State.ATTACK:
                    // Animator의 IsAttack 변수를 true로 설정
                    anim.SetBool(hashAttack, true);
                    break;

                // 사망
                case State.DIE:
                    isDie = true;
                    // 추적 정지
                    agent.isStopped = true;
                    // 사망 애니메이션 실행
                    anim.SetTrigger(hashDie);
                    // 몬스터의 Collider 컴포넌트 비활성화
                    GetComponent<CapsuleCollider>().enabled = false;
                    // 일정 시간 대기 후 오브젝트 풀링으로 환원
                    yield return new WaitForSeconds(3.0f);
                    // 사망 후 다시 사용할 때를 위해 hp 값 초기화
                    hp = 100;
                    isDie = false;
                    // 몬스터의 Collider 컴포넌트 활성화
                    GetComponent<CapsuleCollider>().enabled = true;
                    // 몬스터를 비활성화
                    this.gameObject.SetActive(false);
                    break;
            }
            yield return new WaitForSeconds(0.3f);
        }
    }

    void OnDrawGizmos()
    {
        // 추적 사정거리 표시
        if (state == State.TRACE)
        {
            Gizmos.color = Color.blue;
            Gizmos.DrawWireSphere(transform.position, traceDist);
        }
        // 공격 사정거리 표시
        if (state == State.ATTACK)
        {
            Gizmos.color = Color.red;
            Gizmos.DrawWireSphere(transform.position, attackDist);
        }
    }
    void OnCollisionEnter(Collision coll)
    {
        if (coll.collider.CompareTag("BULLET"))
        {
            // 충돌한 총알을 삭제
            Destroy(coll.gameObject);
            // 피격 리액션 애니메이션 실행
            anim.SetTrigger(hashHit);


            // 총알의 충돌 지점
            Vector3 pos = coll.GetContact(0).point;
            // 총알의 충돌 지점의 법선 벡터
            Quaternion rot = Quaternion.LookRotation(-coll.GetContact(0).normal);
            // 혈흔 효과를 생성하는 함수 호출
            ShowBloodEffect(pos, rot);
            // 몬스터의 hp 차감
            hp -= 10;
            if (hp <= 0)
            {
                state = State.DIE;
            }
        }
    }
    // 자기 자신 충돌 감지를 위해서 추가
    void OnTriggerEnter(Collider coll)
    {
        Debug.Log(coll.gameObject.name);
    }
    void ShowBloodEffect(Vector3 pos, Quaternion rot)
    {
        // 혈흔 효과 생성
        GameObject blood = Instantiate<GameObject>(bloodEffect, pos, rot, monsterTr);
        Destroy(blood, 1.0f);
    }
    void OnPlayerDie()
    {
        // 몬스터의 상태를 체크하는 코루틴 함수를 모두 정지시킴
        StopAllCoroutines();

        // 추적을 정지하고 애니메이션을 수행
        agent.isStopped = true;
        // 스피드를 위해서 추가해야 하는 코드
        anim.SetFloat(hashSpeed, Random.Range(0.8f, 1.2f));
        anim.SetTrigger(hashPlayerDie);
    }
}
```

스코어 UI 구현

게임 진행 중 몬스터를 죽였을 때 스코어를 50점씩 증가시키는 로직을 GameManager에서 구현한다
다음과 같이 GameManager 스크립트에 추가한다

네임 스페이스 추가

```
// TextMesh Pro 관련 컴포넌트에 접근하기 위해 선언
using TMPro;
```

변수 추가

```
// 스코어 텍스트를 연결할 변수
public TMP_Text scoreText;
// 누적 점수를 기록하기 위한 변수
private int totScore = 0;
```

Start 함수에 추가

```
// 스코어 점수 출력
DisplayScore(0);
```

함수 추가

```
// 점수를 누적하고 출력하는 함수
public void DisplayScore(int score)
{
    totScore += score;
    scoreText.text = $"<color=#00ff00>SCORE :</color> <color=#ff0000>{totScore:#,##0}</color>";
}
```

스코어 UI 구현

이제 MonsterCtrl 스크립트를 수정해 몬스터가 죽었을 때 GameManager의 DisplayScore를 호출하게 한다
MonsterCtrl 스크립트의 OnCollisionEnter 함수를 다음과 같이 수정한다

```
void OnCollisionEnter(Collision coll)
    {
        if (coll.collider.CompareTag("BULLET"))
        {
            // 총알을 삭제
            Destroy(coll.gameObject);
            // 피격 리액션 애니메이션 실행
            anim.SetTrigger(hashHit);


            // 총알의 충돌 지점
            Vector3 pos = coll.GetContact(0).point;
            // 총알의 충돌 지점의 법선 벡터
            Quaternion rot = Quaternion.LookRotation(-coll.GetContact(0).normal);
            // 혈흔 효과를 생성하는 함수 호출
            ShowBloodEffect(pos, rot);

            // 몬스터의 hp 차감
            hp -= 10;

            if (hp <= 0)
            {
                state = State.DIE;
                //몬스터가 사망했을 때 50점을 추가
                GameManager.instance.DisplayScore(50);
            }
        }
    }
```

스코어 UI 구현

```
void Start()
{
    // 몬스터 오브젝트 풀 생성
    CreateMonsterPool();

    // SpawnPointGroup 게임오브젝트의 Transform 컴포넌트 추출
    Transform spawnPointGroup = GameObject.Find("SpawnPointGroup")?.transform;

    // SpawnPointGroup 하위에 있는 모든 차일드 게임오브젝트의 Transform 컴포넌트 추출
    foreach (Transform point in spawnPointGroup)
    {
        points.Add(point);
    }

    // 일정한 시간 간격으로 함수를 호출
    InvokeRepeating("CreateMonster", 2.0f, createTime);

    totScore = PlayerPrefs.GetInt("TOT_SCORE", 0);

    // 스코어 점수 출력
    DisplayScore(0);
}
```

```
public void DisplayScore(int score)
{
    totScore += score;
    scoreText.text = $"<color=#00ff00>SCORE :</color> <color=#ff0000>{totScore:#,##0}</color>";
    // 스코어 저장
    PlayerPrefs.SetInt("TOT_SCORE", totScore);
}
```

레이캐스트 활용

레이캐스트는 비단 발사 로직뿐만 아니라 감지 센서 역할과 클릭 후 이동, 또는 회전하는 데도 활용된다
예를 들어, 디아블로나 리니지 같은 전통적인 쿼터뷰(Quarter View) 방식의 게임에서는 플레이어를 마우스 왼쪽 클릭으로 조작해 이동시킬 때 실제로는 마우스 포인트 위치로 레이캐스트해서 3차원 좌푯값을 읽어온 후 해당 좌표로 이동시킨다

DrawRay

레이캐스트는 씬 뷰에서 시각적으로 표시되지 않기 때문에 개발할 때는 DrawRay 함수를 이용해 시각적으로 표시하고 개발을 진행해야 한다
FireCtrl 스크립트의 Update 함수에 다음과 같이 Debug.DrawRay 함수를 추가한다

```
void Update()
    {
        // Ray를 시각적으로 표시하기 위해 사용
        Debug.DrawRay(firePos.position, firePos.forward * 10.0f, Color.green);

        // 마우스 왼쪽 버튼을 클릭했을 때 Fire 함수 호출
        if (Input.GetMouseButtonDown(0))
        {
            Fire();
        }
    }
```

Raycast, RaycastHit

이제 물리적인 총알의 용도는 단순히 시각적인 역할을 할 뿐이고, 실제 몬스터는 Ray에 맞았을 때 데미지를 입도록 로직을 수정해보자
먼저 Ray에 맞아서 사망하는 결과를 확인하기 위해서 Instantiate 함수로 총알을 생성하는 로직은 잠시 주석 처리한다
FireCtrl 스크립트를 다음과 같이 수정한다

변수 추가

```
// Raycast 결괏값을 저장하기 위한 구조체 선언
private RaycastHit hit;
```

Update함수에 추가

```
void Update()
    {
        // Ray를 시각적으로 표시하기 위해 사용
        Debug.DrawRay(firePos.position, firePos.forward * 10.0f, Color.green);

        // 마우스 왼쪽 버튼을 클릭했을 때 Fire 함수 호출
        if (Input.GetMouseButtonDown(0))
        {
            Fire();

            // Ray를 발사
            if (Physics.Raycast(firePos.position,    // 광선의 발사 원점
                                firePos.forward,     // 광선의 발사 방향
                                out hit,             // 광선에 맞은 결과 데이터
                                10.0f,               // 광선의 거리
                                1 << 6))             // 감지하는 범위인 레이어 마스크
            {
                Debug.Log($"Hit={hit.transform.name}");
            }
        }
    }
```

Fire 함수에 주석 처리 실행

```
void Fire()
    {
        // Bullet 프리팹을 동적으로 생성(생성할 객체, 위치, 회전)
        //Instantiate(bullet, firePos.position, firePos.rotation);
        // 총소리 발생
        audio.PlayOneShot(fireSfx, 1.0f);
        // 총구 화염 효과 코루틴 함수 호출
        StartCoroutine(ShowMuzzleFlash());
    }
```

Raycast, RaycastHit

실제로 Bullet 모델을 발사해 충돌을 발생시킨 이전 로직은 MonsterCtrl스크립트의 OnCollisionEnter 함수에서 피격 시 혈흔 효과와 hp 수치를 감소시켰다
그러나 레이캐스트로 변경한 방식은 실제 충돌이 발생하지 않아 OnCollisionEnter콜백 함수는 당연히 발생하지 않는다
대신 총을 쏜 플레이어가 몬스터에게 “너, 총 맞았으니까 Hp 줄이고 피 흘려라 " 라고 알려줘야 한다
MonsterCtrl스크립트에 다음과 같이 OnCollisionEnter함수를 수정하고 OnDamage함수를 추가한다

```
void OnCollisionEnter(Collision coll)
    {
        if (coll.collider.CompareTag("BULLET"))
        {
            // 충돌한 총알을 삭제
            Destroy(coll.gameObject);
        }
    }
```

```
// 레이캐스트를 사용해 데미지를 입히는 로직
    public void OnDamage(Vector3 pos, Vector3 normal)
    {
        // 피격 리액션 애니메이션 실행
        anim.SetTrigger(hashHit);
        Quaternion rot = Quaternion.LookRotation(normal);

        // 혈흔 효과를 생성하는 함수 호출
        ShowBloodEffect(pos, rot);

        // 몬스터의 hp 차감
        hp -= 30;
        if (hp <= 0)
        {
            state = State.DIE;
            // 몬스터가 사망했을 때 50점을 추가
            GameManager.instance.DisplayScore(50);
        }
    }
```

Raycast, RaycastHit

이제 FireCtrl스크립트에서 MonsterCtrl스크립트의 OnDamage 함수를 호출한다
FireCtrl스크립트의 Update 함수와 Fire 함수를 다음과 같이 수정한다

```
void Update()
    {
        // Ray를 시각적으로 표시하기 위해 사용
        Debug.DrawRay(firePos.position, firePos.forward * 10.0f, Color.green);

        // 마우스 왼쪽 버튼을 클릭했을 때 Fire 함수 호출
        if (Input.GetMouseButtonDown(0))
        {
            Fire();

            // Ray를 발사
            if (Physics.Raycast(firePos.position,   // 광선의 발사 원점
                                firePos.forward,    // 광선의 발사 방향
                                out hit,            // 광선에 맞은 결과 데이터
                                10.0f,              // 광선의 거리
                                1 << 6))            // 감지하는 범위인 레이어 마스크
            {
                Debug.Log($"Hit={hit.transform.name}");
                hit.transform.GetComponent<MonsterCtrl>()?.OnDamage(hit.point, hit.normal);
            }
        }
    }

    void Fire()
    {
        // Bullet 프리팹을 동적으로 생성(생성할 객체, 위치, 회전)
        Instantiate(bullet, firePos.position, firePos.rotation);
        // 총소리 발생
        audio.PlayOneShot(fireSfx, 1.0f);
        // 총구 화염 효과 코루틴 함수 호출
        StartCoroutine(ShowMuzzleFlash());
    }
```

이제 게임을 실행해 총을 발사하면 물리적인 Bullet에 맞았을 때와 같은 효과가 발생하는 것을 확인할 수 있다

MonsterCtrl스크립트의 OnCollisionEnter 함수에서 몬스터에 데미지를 입히는 로직은 모두 OnDamage 함수로 이동했기 때문에 물리적인 총알에 맞아도 데미지에는 영향을 주지 않는다

따라서 FireCtrl스크립트의 Fire 함수에서 주석 처리했던 Instantiate 함수는 다시 주석을 해제한다 즉, 총알은 시각적인 효과로만 사용하는 것이다

실행하면 총알이 발사되지만 실제 몬스터에게 데미지를 주는 로직은 레이캐스트를 통해서 전달된다

자연스러운 회전 처리

현재 몬스터의 이동과 회전은 NavMeshAgent에서 처리하고 있다
이동과는 다르게 회전 로직은 이동하면서 회전하기 때문에 부자연스러운 동작을 볼 수 있다
따라서 빠르게 회전 처리하고 이동하도록 스크립트에서 직접 회전 로직을 구현해본다
MonsterCtrl스크립트의 Awake함수를 수정하고 Update함수를 다음과 같이 추가한다

```
void Awake()
    {
        // 몬스터의 Transform 할당
        monsterTr = GetComponent<Transform>();

        // 추적 대상인 Player의 Transform 할당
        playerTr = GameObject.FindWithTag("PLAYER").GetComponent<Transform>();

        // NavMeshAgent 컴포넌트 할당
        agent = GetComponent<NavMeshAgent>();

        // NavMeshAgent의 자동 회전 기능 비활성화
        agent.updateRotation = false;

        // Animator 컴포넌트 할당
        anim = GetComponent<Animator>();

        // BloodSprayEffect 프리팹 로드
        bloodEffect = Resources.Load<GameObject>("BloodSprayEffect");
    }
```

```
void Update()
    {
        // 목적지까지 남은 거리로 회전 여부 판단
        if (agent.remainingDistance >= 2.0f)
        {
            // 에이전트의 이동 방향
            Vector3 direction = agent.desiredVelocity;

            if (direction.sqrMagnitude >= 0.1f * 0.1f)
            {
                // 회전 각도(쿼터니언) 산출
                Quaternion rot = Quaternion.LookRotation(direction);
                // 구면 선형보간 함수로 부드러운 회전 처리
                monsterTr.rotation = Quaternion.Slerp(monsterTr.rotation,
                                                      rot,
                                                      Time.deltaTime * 10.0f);
            }
        }
    }
```

Scene 병합

게임 개발 시 처음 진입하는 화면은 게임 동영상 또는 Cut-Scene애니메이션 화면을 제외하고는 대부분 메인 메뉴가 있는 화면이 메인 화면이 될 것이다
메인 화면은 앞서 만든 Main씬을 사용하고, Main씬에 있는 START버튼을 클릭하면 Level_01씬과 Play씬 두개의 씬을 합쳐서 보여주는 로직으로 구현해본다
Main씬을 열고 UIManager스크립트를 다음과 같이 수정한다

```
public class UIManager : MonoBehaviour
{
    // 버튼을 연결할 변수
    public Button startButton;
    public Button optionButton;
    public Button shopButton;

    private UnityAction action;

    void Start()
    {

        // UnityAction을 사용한 이벤트 연결 방식
        action = () => OnButtonClick(startButton.name);
        startButton.onClick.AddListener(action);

        // 무명 메서드를 활용한 이벤트 연결 방식
        optionButton.onClick.AddListener(delegate { OnButtonClick(optionButton.name); });

        // 람다식을 활용한 이벤트 연결 방식
        shopButton.onClick.AddListener(() => OnButtonClick(shopButton.name));
    }

    public void OnButtonClick(string msg)
    {
        Debug.Log($"Click Button : {msg}");
    }

    public void OnStartClick()
    {
        SceneManager.LoadScene("Level_01");
        SceneManager.LoadScene("Play", LoadSceneMode.Additive);
    }
}
```

씬 분리

OnStartClick은 START버튼을 클릭했을 때 호출할 함수이므로 Start 함수의 이벤트에 연결한다
기존에 연결했던 OnButtonClick함수를 OnStartClick함수로 교체한다

```
void Start()
    {

        // UnityAction을 사용한 이벤트 연결 방식
        action = () => OnStartClick();
        startButton.onClick.AddListener(action);

        // 무명 메서드를 활용한 이벤트 연결 방식
        optionButton.onClick.AddListener(delegate { OnButtonClick(optionButton.name); });

        // 람다식을 활용한 이벤트 연결 방식
        shopButton.onClick.AddListener(() => OnButtonClick(shopButton.name));
    }
```

끝

다들 수고 했습니다