UNITY
-CAHPTER7-

SOUL SEEK

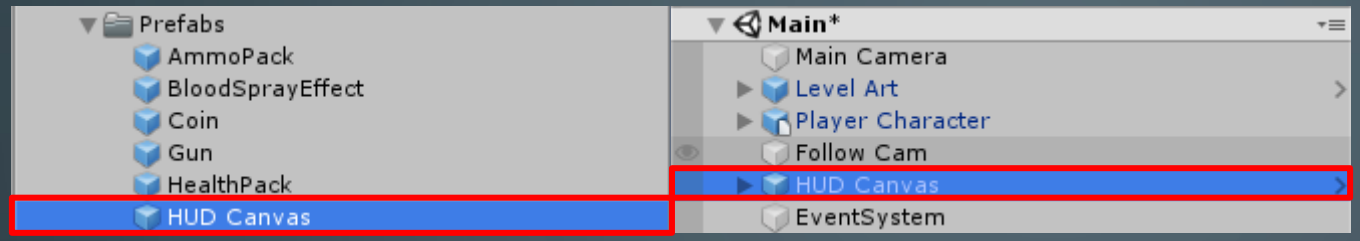
목차

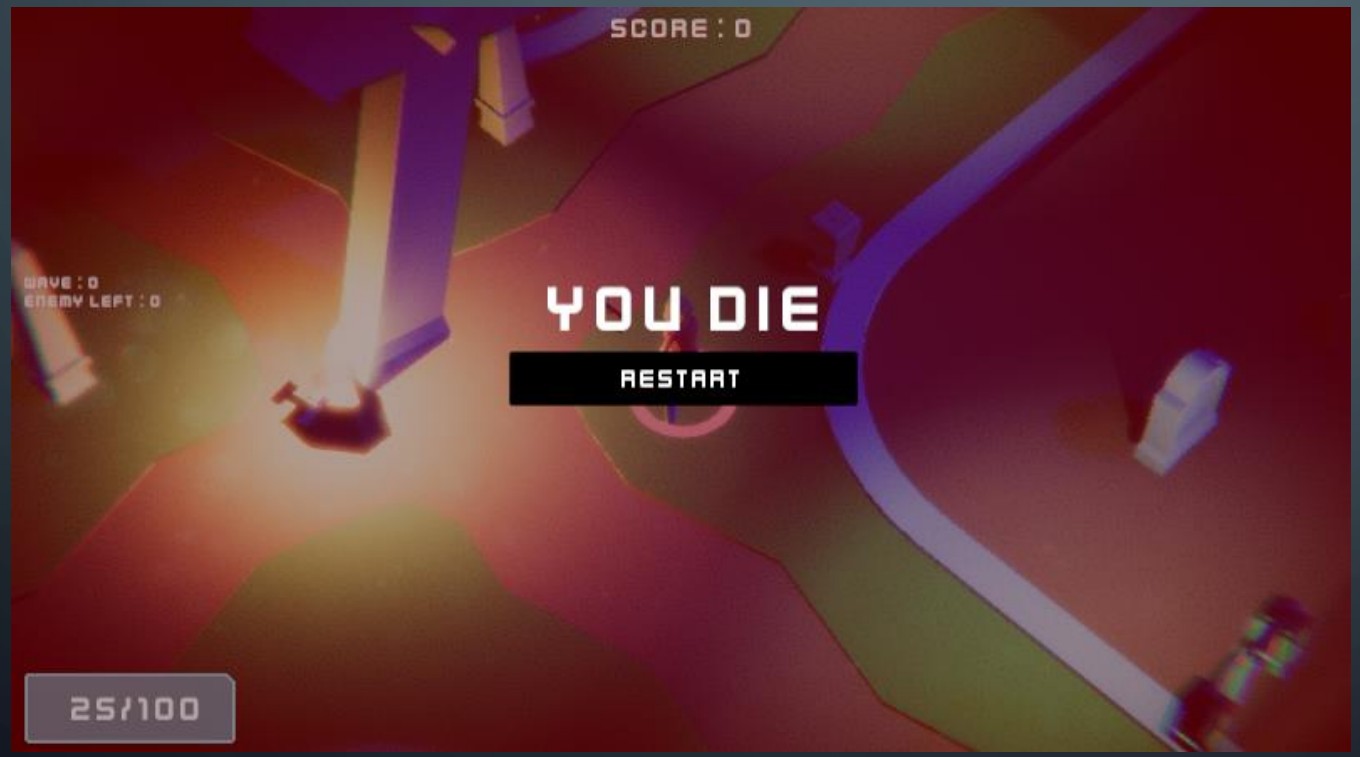
1. HUD Canvas와 UI Manager
2. GameManager
3. EnemySpawner와 Item Creator
4. Post Processing

HUD CANVAS와 UI MANAGER

1. HUD CANVAS

남은 탄알과 적 웨이브, 점수, **GameOver** 등을 표시할 **HUD Canvas**를 추가한다.





HUD Canvas는 여러 개의 자식 **GameObject**를 가지고 있다.

Ammo Display : 남은 탄알을 표시하는 **Background**
→**Ammo Text :** 탄알을 표시

Score Text : 점수를 표시

Enemy Wave Text : 현재 적 **Wave**와 남은 적 수를 표시

Gameover UI : Gameover 시 활성화되는 **Panel**
→**Gameover Text :** 게임오버
→**Restart Button :** 게임 재시작
→**Text :** 버튼의 텍스트

2. UI MANAGER

UI Manager로 HUD Canvas를 관리하는 이유

- **HUD Canvas GameObject**는 여러 **UI** 요소를 자식으로 가지고 있다.
- **HUD Canvas**의 **UI** 요소들을 사용하려는 **Script**가 **HUD Canvas**의 **UI** 요소를 직접 접근하고 사용한 다면 **UI**구현을 유연하게 변경하기 힘들다. **HUD Canvas**의 **UI** 구현이 변경되면 각각의 **UI** 요소를 참조 하고있던 여러 **Script**의 구현도 함께 변경해야 하는 상황이 생길 수 있기 때문에 **UI Manager**를 **UI** 관리 용 **Script**로 만들어서 **HUD Canvas**의 자식으로 있는 개별 **UI**를 관리하는 코드를 구현하면 된다.

UI Manager의 기능

- **Singleton** 디자인 패턴으로 관리
- **HUD Canvas**의 **UI** 요소에 즉시 접근할 수 있는 통로
- **HUD Canvas**의 **UI** 관련 구현을 모아두는 **Script**

싱글톤 프로퍼티

```
private static UIManager m_instance; // 싱글톤이 할당될 변수
public static UIManager instance // 싱글톤 접근용 프로퍼티
{
    get
    {
        if (m_instance == null)
        {
            m_instance = FindObjectOfType<UIManager>();
        }

        return m_instance;
    }
}
```

- **Instance**는 **public get**만 존재하는 프로퍼티이며, 외부 **Script**에서 **UIManager.instance**를 최초로 접근할 때는 **m_instance**에 아직 아무런 값도 할당되지 않아 **m_instance**의 값이 **null**이 된다.
- **Instance**의 **get**이 실행되면서 **m_instance**에 할당된 값이 없다면 아래 **if**문 블록이 실행되어 **Sce ne**에 존재하는 **UIManager Type**의 오브젝트를 하나 찾아 **m_instance**에 할당한다. 그리고 나서 **m_instance**를 반환한다.

2. UI MANAGER

UI Manager의 Field

HUD Canvas GameObject의 자식으로 있는 **UI GameObject**를 할당해 넣은 변수가 선언되어 있다.

```
public Text ammoText; // 탄약 표시용 텍스트
public Text scoreText; // 점수 표시용 텍스트
public Text waveText; // 적 웨이브 표시용 텍스트
public GameObject gameoverUI; // 게임 오버시 활성화할 UI
```

- **Text ammoText : Ammo Text GameObject**의 **Text Component**가 할당.
- **Text scoreText : Score Text GameObject**의 **Text Component**가 할당.
- **Text waveText : Enemy Wave Text GameObjece**의 **Text Component**가 할당.
- **GameObject gameoverUI : Gameover UI GameObject**가 할당.

UI 갱신 Method

UI GameObject가 표시하는 내용을 갱신하는 **Method**를 선언해 두었다.

```
// 탄약 텍스트 갱신
public void UpdateAmmoText(int magAmmo, int remainAmmo)...

// 점수 텍스트 갱신
public void UpdateScoreText(int newScore)...

// 적 웨이브 텍스트 갱신
public void UpdateWaveText(int waves, int count)...

// 게임 오버 UI 활성화
public void SetActiveGameoverUI(bool active)...
```

탄알 표시 **UI** 갱신
탄창의 탄알 **magAmmo,** 남은 탄알 **remain Ammo**를 입력받아 표시

점수 표시 **UI** 갱신**,** 표시할 점수 **newScore**를 입력받음

적 웨이브 정보 **UI**갱신
현재 적 웨이브와 남아 있는 적 수를 입력받아 표시

게임오버 **UI** 패널을 활성 / 비활성화

2. UI MANAGER

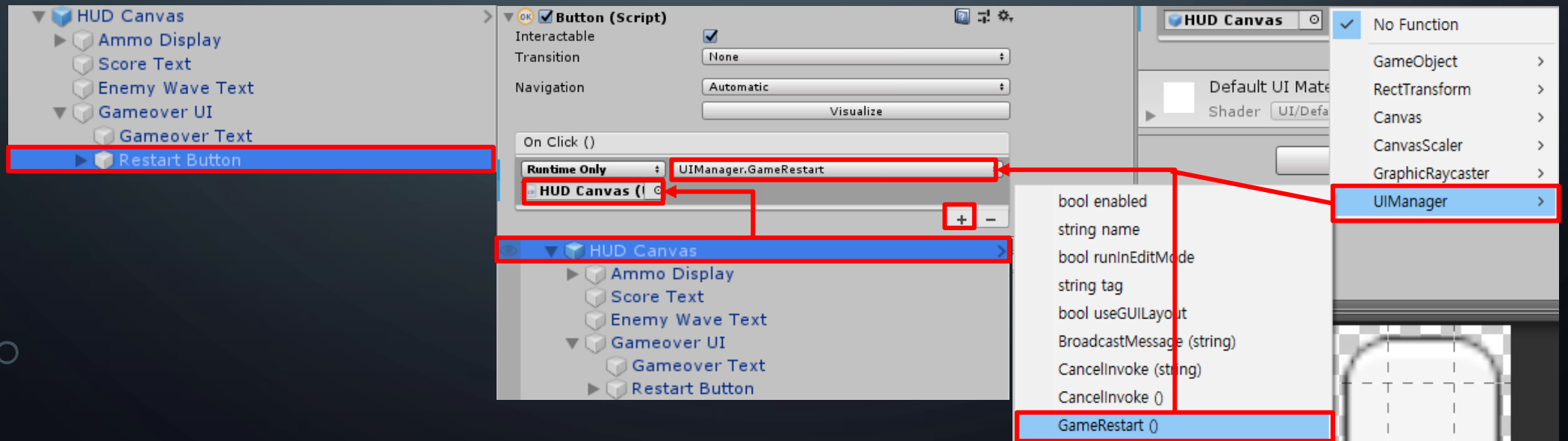
GameRestart() Method

게임을 재시작하는 **Method**
HUD Canvas의 재시작 버튼에 할당해서 사용한다**.**

```
// 게임 재시작
public void GameRestart()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
```

재시작버튼 설정

1. 하이어라키 창에서 **Restart Button GameObject**을 선택하고 **Button Component**의 **On Click()**리스트의 추가 버튼을 클릭한 후 생성된 슬롯에 **HUD Canvas GameObject**를 **Drag&Drop**한다**.**
2. 이벤트 리스너로 **UIManager.GameRestart**를 등록**(No Function > UIManager > GameRestart()**클릭**)**

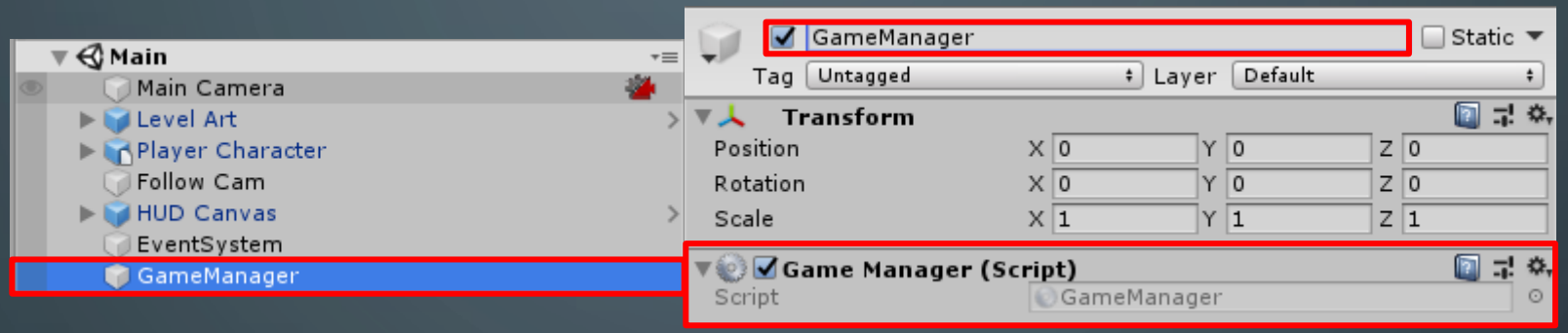


GAMEMANAGER

1. GAMEMANAGER

GameManager 추가

1. **Empty GameObject** 생성**(Create > Create Empty)**한 후 **GameManager**로 이름을 변경하고 **GameManager GameObject**에 **GameManager Script** 추가**(Script** 폴더의 **GameManager Scr ipt**를 **GameManager GameObject**에 추가**)**



GameManager의 역할

- 싱글턴으로 존재**,** 점수 관리**,** 게임오버 상태 관리**, UIManager**를 이용해 점수와 게임오버 **UI** 갱신

싱글턴 프로퍼티

- 다른 **Script**들에서 즉시 접근할 수 있도록 만드는 싱글턴 프로퍼티

```
private static GameManager m_instance; // 싱글톤이 할당될 static 변수
public static GameManager instance // 싱글톤 접근용 프로퍼티
{
    get
    {
        // 만약 싱글톤 변수에 아직 오브젝트가 할당되지 않았다면
        if (m_instance == null)
        {
            // 씬에서 GameManager 오브젝트를 찾아 할당
            m_instance = FindObjectOfType<GameManager>();
        }

        // 싱글톤 오브젝트를 반환
        return m_instance;
    }
}
```

1. GAMEMANAGER

GameManager의 Field

- 게임의 상태를 나타내는 변수와 프로퍼티

```
private int score = 0; // 현재 게임 점수
public bool isGameover { get; private set; } // 게임 오버 상태
```

- **isGameover**는 **public get, private set**으로 선언된 프로퍼티이기 때문에 **GameManager** 외부에서는 값을 읽을 수 만 있고 변경할 수 없다.

Awake() Method

- **Scene**에서 둘 이상의 **GameManager Type**의 오브젝트가 존재하지 못하도록 막는다.

```
private void Awake()
{
    // 씬에 싱글톤 오브젝트가 된 다른 GameManager 오브젝트가 있다면
    if (instance != this)
    {
        // 자신을 파괴
        Destroy(gameObject);
    }
}
```

- **Awake()**의 **if** 문 블록은 싱글턴 프로퍼티 **instance**를 이용해 접근한 싱글턴 오브젝트가 자기 자신**(this)**이 아니라면 **(**이미 다른 **GameManager Type** 오브젝트가 싱글턴으로 존재하고 있다면**)** 자신의 **GameObject**를 파괴한다 싱글턴 오브젝트는 하나만 존재해야 하기 때문이다.

1. GAMEMANAGER

Start() Method

- **Player**가 사망하면 게임오버 처리를 실행하도록 한다.

```
private void Start()
{
    // 플레이어 캐릭터의 사망 이벤트 발생시 게임 오버
    FindObjectOfType<PlayerHealth>().onDeath += EndGame;
}
```

- **Scene**에서 **PlayerHealth Type**의 오브젝트를 찾고, 해당 오브젝트의 **onDeath** 이벤트를 **EndGame() Method**가 구독하는 처리이다. 따라서 플레이어 캐릭터가 사망하면서 **onDeath** 이벤트가 발동될 때 **onDeath**를 구족 중인 **End Game() Method**가 함께 실행되어 게임오버 처리가 실행된다.

AddScore() Method

- 점수를 입력 받아 현재 점수에 추가하고 점수 **UI**를 갱신한다.

```
// 점수를 추가하고 UI 갱신
public void AddScore(int newScore)
{
    // 게임 오버가 아닌 상태에서만 점수 증가 가능
    if (!isGameover)
    {
        // 점수 추가
        score += newScore;
        // 점수 UI 텍스트 갱신
        UIManager.instance.UpdateScoreText(score);
    }
}
```

- **If(!isGameover)**를 사용해 게임오버가 아닌 상태에서만 점수를 추가하고 **UI** 텍스트를 갱신한다. 또한 **GameMa nager Script**에서 **UI GameObject**를 직접 수정하지 않고, **UIManager**싱글턴을 거쳐 점수 **UI**를 갱신한다.
  →**UIManager** 덕분에 **GameManager Script**에서는 점수 **UI**가 갱신되는 처리의 세부구현을 신경 쓸 필요가 없으므로 깔끔한 코드 를 작성할 수 있다.

1. GAMEMANAGER

EndGame() Method

현재 게임 상태를 게임오버 상태로 전환하고 게임 오버 **UI**를 활성화 한다.

```
// 게임 오버 처리
public void EndGame()
{
    // 게임 오버 상태를 참으로 변경
    isGameover = true;
    // 게임 오버 UI를 활성화
    UIManager.instance.SetActiveGameoverUI(true);
}
```

AddScore()처럼 **UIManager** 싱글턴을 거쳐 **UI**를 갱신한다.

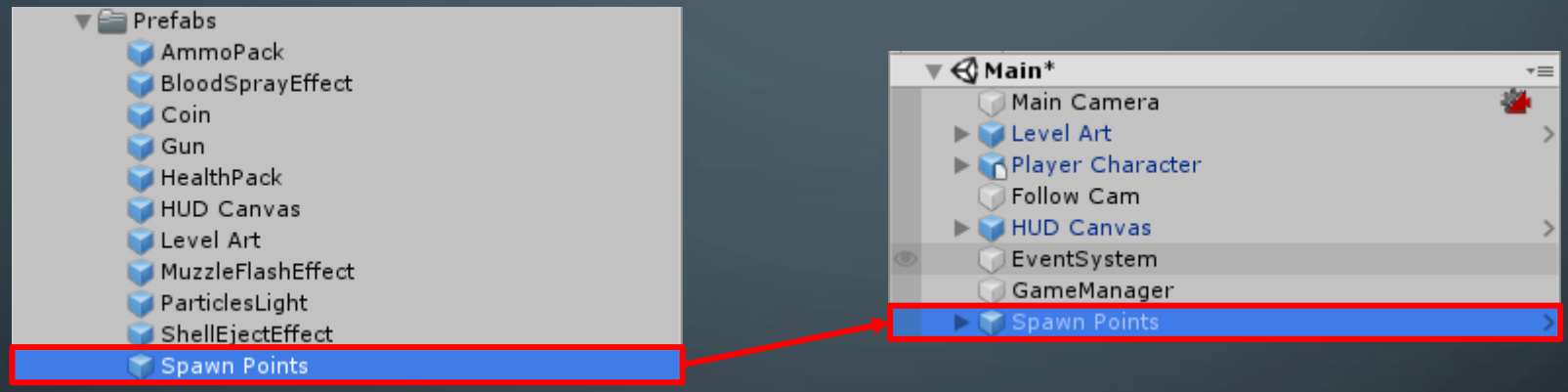
ENEMY SPAWNER와 ITEM CREATOR

1. ENEMY SPAWNER

EnemySpwaner의 기능 설정

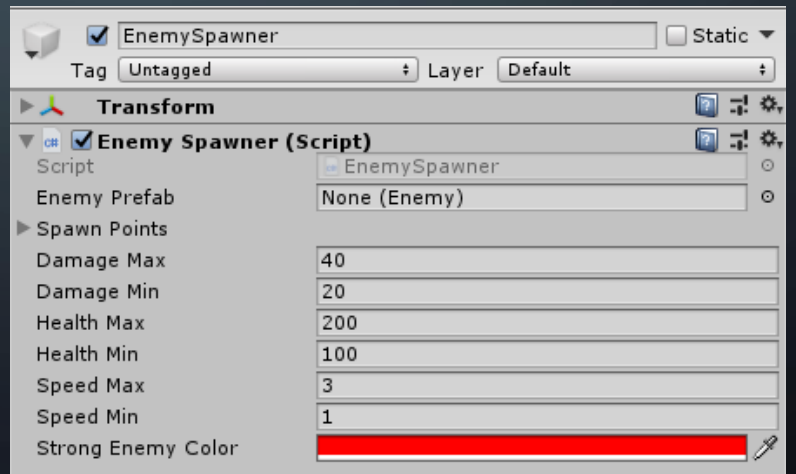
1. 새로운 웨이브가 시작될 때 적을 한꺼번에 생성
2. 현재 웨이브의 적이 모두 사망해야 다음 웨이브로 넘어간다
3. 웨이브가 증가할 때마다 한 번에 생성되는 적 수 증가
4. 적을 생성할 때 전체 능력치를 **0%**에서 **100%** 사이에서 랜덤 설정**,** 게임오버 시 적 생성 중단

생성위치 추가



적 생성기 추가

1. 빈 게임 오브젝트 생성하고 **Enemy Spawner**로 이름 변경
2. **Enemy Spawner GameObject**에 **Enemy Spawner Script** 추가



1. ENEMY SPAWNER

EnemySpawner의 Field

- 실시간으로 생성할 적의 원본 **Prefab**을 할당할 **Enemy Type**의 변수 선언
- **spawnPoints**는 적 생성 위치로 사용할 트랜스폼을 저장하는 배열

```
public Enemy enemyPrefab; // 생성할 적 AI

public Transform[] spawnPoints; // 적 AI를 소환할 위치들
```

- 생성한 적의 능력치(공격력, 체력, 이동 속도)를 설정하는 데 사용할 최소값과 최대값 변수가 나열

```
public float damageMax = 40f; // 최대 공격력
public float damageMin = 20f; // 최소 공격력

public float healthMax = 200f; // 최대 체력
public float healthMin = 100f; // 최소 체력

public float speedMax = 3f; // 최대 속도
public float speedMin = 1f; // 최소 속도
```

- **strongEnemyColor**는 생성한 적의 능력치가 최대값일 때 사용할 피부색을 나타내는 변수
- **EnemySpawner Script**로 생성된 적의 능력치는 **0%**(최소)에서 **100%**(최대) 사이에서 랜덤 결정된다. 적의 능력치가 **100%**에 가까울수록 최대값에 가까운 수치가 능력치로 사용되며, 피부색도 **strongEnemyColor**에 할당된 컬러에 가까 워지게 된다.

```
public Color strongEnemyColor = Color.red; // 강한 적 AI가 가지게 될 피부색

private List<Enemy> enemies = new List<Enemy>(); // 생성된 적들을 담는 리스트
private int wave; // 현재 웨이브
```

- 생성한 적들을 등록하고 추적하는 데 사용할 리스트 **enemies**가 선언되어 있다.
- **Enemies**는 현재 살아 있는 적 수를 파악하는 데 사용, **wave**는 현재 적 생성 웨이브를 나타낸다. 현재 웨이브의 모든 적을 제거할 때마다 웨이브가 **1**씩 증가하며, 웨이브 값이 클수록 한 번에 생성되는 적 수도 많아진다.

1. ENEMY SPAWNER

Update() Method

- 매 프레임마다 조건을 검사하고 적 생성 웨이브를 실행한다.

```
private void Update()
{
    // 게임 오버 상태일때는 생성하지 않음
    if (GameManager.instance != null && GameManager.instance.isGameover)
    {
        return;
    }

    // 적을 모두 물리친 경우 다음 스폰 실행
    if (enemies.Count <= 0)
    {
        SpawnWave();
    }

    // UI 갱신
    UpdateUI();
}
```

- **GameManager** 싱글턴이 존재하며**, GameManager** 싱글턴의 **isGameover**가 **true**인 게임오버 상태라면 처리를 더 이상 진행하지 않고 즉시 **Update() Method**를 종료
- 게임오버 상태가 아니라면 처리가 계속 진행되어 적 생성 웨이브를 실행할 조건을 검사**,** 만약 적 리스트 **enemies**에 등록 된 적 수가 **0**보다 작거나 같은 경우**,** 즉 **Scene**에 적이 남아 있지 않다면 적 생성 웨이브 메서드인 **SpawnWave()**를 실 행하려 적을 생성한다**.**
- 모든 조건이 만족하면 **UI**를 갱신하는 **UpdateUI() Method**를 실행한다**.**

1. ENEMY SPAWNER

UpdateUI() Method

- 현재 웨이브 번호와 **Scene**에 남아 있는 적수를 표시하는 **UI**를 갱신한다.

```
// 웨이브 정보를 UI로 표시
private void UpdateUI()
{
    // 현재 웨이브와 남은 적의 수 표시
    UIManager.instance.UpdateWaveText(wave, enemies.Count);
}
```

SpawnWave() Method

- 적 생성 웨이브를 구현하는 **Method,** 적을 실제 생성하는 **Method**는 **SpawnWave()** 다음에 구현할 **CreateEnemy() Method, SpawnWave() Method**는 적을 직접 생성하지 않는 대신**,** 현재 웨이브에 생성할 적 수만큼 **CreateEnemy() Method**를 반복 실행한다.

```
// 현재 웨이브에 맞춰 적을 생성
private void SpawnWave()
{
    //웨이브 카운트 증가
    wave++;

    //현재 웨이브 * 1.5fmf 반올림한 수만큼 적 생성
    int spawnCount = Mathf.RoundToInt(wave * 1.5f);

    //spawnCount만큼 적 생성
    for(int i = 0; i < spawnCount; i++)
    {
        // 적의 세기를 0%dptj 100% 사이에서 랜덤 결정
        float enemyIntensity = Random.Range(0f, 1f);
        // 적 생성 처리 실행
        CreateEnemy(enemyIntensity);
    }
}
```

- 새로운 웨이브를 시작하면서 **wave**를 **1**씩 증가시킨다**.**
- **wave**에 **1.5**를 곱하고 반올림한 값을 생성할 적수인 **spawnCount**의 값으로 사용한다**.**
- 반올림에 사용한 **Mathf.RoundToInt() Method**는 **float** 값을 입력받고 입력값을 반올림한 정수를 반환
- **CreateEnemy() Method**는 적의 강함을 **0 ~ 1**의 값으로 비율적인 값을 받는다**. Random.Range()**로 이 비율을 설정해서 받는다**.**

1. ENEMY SPAWNER

CreateEnemy() Method

생성할 적의 강함 **intensity**를 **0**에서 **1.0**사이의 값으로 입력 받고**, Prefab**으로부터 적을 복제 생성한다**.** 그리고 **Scene**에 총 몇 개의 적이 존재하는지 파악할 수 있도록 생성한 적을 **List**에 등록한다**.**

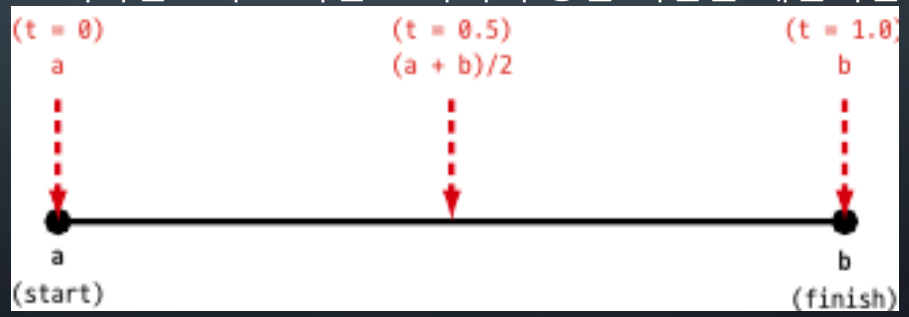
```
// 적을 생성하고 생성한 적에게 추적할 대상을 할당
private void CreateEnemy(float intensity)...
```

생성한 적에 할당할 수치를 최소값과 최대값 사이에서 결정

```
// intensity를 기반으로 적의 능력치 결정
float health = Mathf.Lerp(healthMin, healthMax, intensity);
float damage = Mathf.Lerp(damageMin, damageMax, intensity);
float speed = Mathf.Lerp(speedMin, speedMax, intensity);

// intensity를 기반으로 하얀색과 enemyStrength 사이에서 적의 피부색 결정
Color skinColor = Color.Lerp(Color.white, strongEnemyColor, intensity);
```

적 강함 **intensity**를 기준으로 체력 **health,** 공격력 **damage,** 이동 속도 **speed,** 피부색 **skinColor**의 값을 결정**.**
Mathf.Lerp는 선형보간 **Method**이다
→ 선형 보간은 보간값 **t**를 기준으로 시작점 **a**와 도착점 **b** 사이의 중간 지점을 계산하는 방법이다**.**



시작점 **a**와 도착점 **b**가 있을 때 **t**가 **0**이면 중간 지점은 **a**가 된다**. T**가 **0.5**인 경우 중간 지점은 **a**와 **b**의 정중앙**, t**가 **1.0**인 경우 중간 지점은 **b**가 된다**.**

1. ENEMY SPAWNER

- **float Mathf.Lerp(float a, float b, float t) : t**를 기준으로 **a**와 **b**사이의 중간 값을 반환한다.
- **Color Color.Lerp(Colora, Color b, float t) : t**를 기분으로 **Color a**와 **Color b**를 섞은 **Color**를 반환.

- **health, damage, speed** 값은 **intensity** 값이 **0**에 가까울 수록 **healthMin, damageMin, speedMin**에 가까워지며, **intensity** 값이 **1.0**에 가까울수록 **healthMax, damaMax, speedMax**에 가까워진다.
- **intensity** 값이 **0**에 가까울수록 **skinColor**에 할당될 **Color**는 **Color.white**에 가까워지며, **intensity** 값이 **1.0**에 가까울수록 **strongEnemyColor**에 가까워진다.

```
// 생성할 위치를 랜덤으로 결정
Transform spawnPoint = spawnPoints[Random.Range(0, spawnPoints.Length)];
```

- 생성할 적 수를 결정한 다음에 적을 생성할 위치를 결정한다.
- 생성 위치를 표시할 트랜스폼 **spawnPoint**를 선언하고 **spawnPoint**에 할당된 여러 트랜스폼중 하나를 랜덤 선택해 할당한다. 랜덤 선택에 사용한 배열 순번은 **Random.Range()**를 사용해 **0**부터 배열의 크기인 **spawnPoints.Length** 사이에서 결정

```
// 적 프리팹으로부터 적 생성
Enemy enemy = Instantiate(enemyPrefab, spawnPoint.position, spawnPoint.rotation);

// 생성한 적의 능력치와 추적 대상 설정
enemy.Setup(health, damage, speed, skinColor);

// 생성된 적을 리스트에 추가
enemies.Add(enemy);
```

- 적을 생성하고 생성한 적의 능력치를 설정한다.
- **Instantiate()**로 **enemyPrefab**의 복제봄을 생성하고 **spawnPoint**의 위치와 회전에 배치한다. 그리고 생생된 적 복제본을 **Enemy Type** 변수 **enemy**로 받아 **Setup() Method**를 실행하고, 초기값을 설정한다.
- 초기값이 설정된 **enemy**를 추가한다.

1. ENEMY SPAWNER

- 생성된 적의 **onDeath** 이벤트에 등록한다.
  →**enemies.Remove(enemy) :** 자신을 리스트에서 제거
  →**Destroy(enemy.gameObject, 10f) : 10**초 뒤에 자신의 게임 오브젝트 파괴
  →**GameManager.instance.AddScore(100) :** 게임 점수를 **100**점 증가

```
// 적의 onDeath 이벤트에 익명 메서드 등록
// 사망한 적을 리스트에서 제거
enemy.onDeath += () => enemies.Remove(enemy);
// 사망한 적을 10초 뒤에 파괴
enemy.onDeath += () => Destroy(enemy.gameObject, 10f);
// 적 사망 시 점수 상승
enemy.onDeath += () => GameManager.instance.AddScore(100);
```

- 사망한 적을 적 리스트에서 빼고, 적 시체가 계속 늘어나지 않도록 사망한 적 게임 오브젝트를 **10**초 뒤에 파괴하고, 게임 점수를 **100**점 증가시킨다.
- 추가된 **Method**들은 **EnemySpawner Class**에 없다.
  →익명함수로 만들어져서 이벤트에 등록할 수 있다.

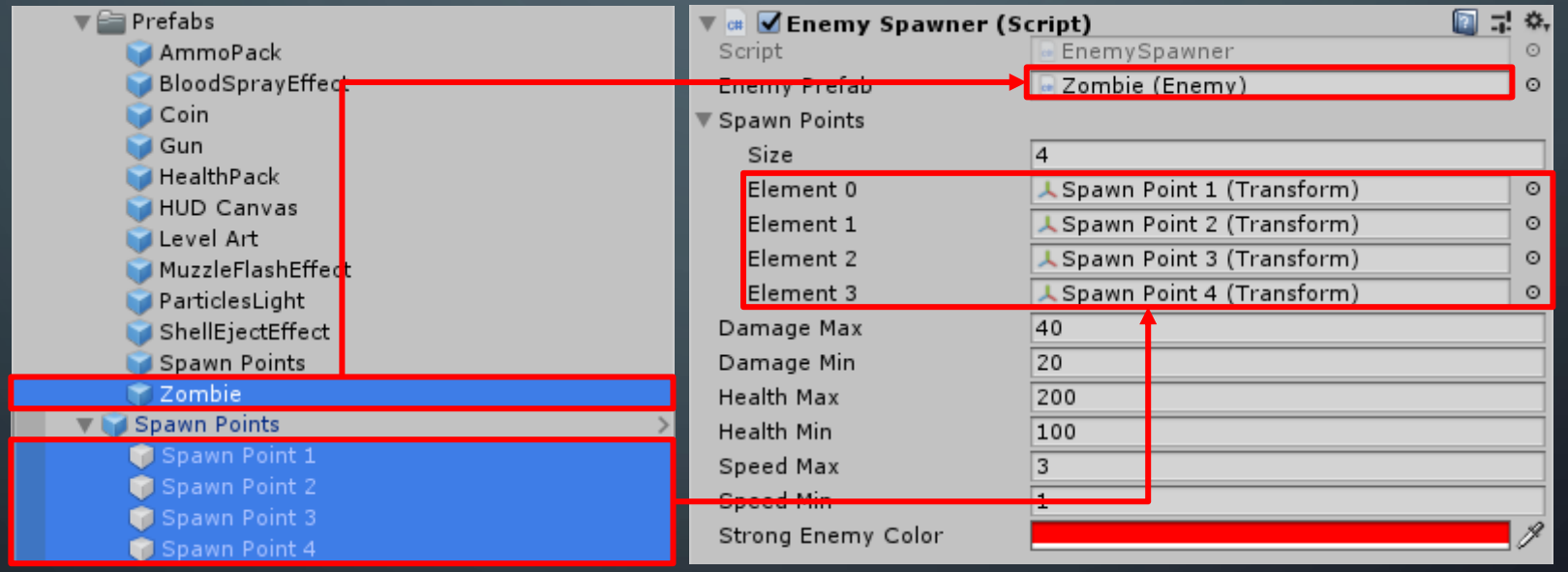
익명 함수와 람다식

- 미리 정의하지 않고, 실행 중인 코드 블록 내부에서 즉석 생성할 수 있는 **Method**
- 실시간으로 생성할 수 있으며, 변수에 저장할 수 있는 값이나 오브젝트로 취급되며, 생성된 익명 함수는 **Delegate Type** 의 변수에 저장할 수 있다.
  →미리 정의하지 않고 대부분 일회용으로 실시간 생성해서 사용하기 때문에 외부에서 따로 지칭할 수 있는 이름을 가지고 있지 않다.
- 간단한 메소드같은 경우에는 직접적인 함수 구현없이 실시간으로 생성해서 등록할 수 있다.

1. ENEMY SPAWNER

EnemySpawner Component 설정

1. **EnemySpawner Component**의 **Enemy Prefab**에 **Prefabs** 폴더의 **Zombie Prefab** 할당
2. **SpawnPoints Field**의 각 원소에 **Spawn Point 1, 2, 3, 4** 할당



2. ITEM SPAWNER

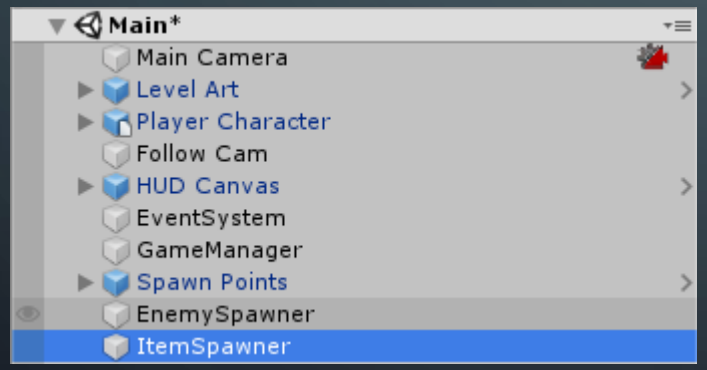
- 플레이어가 사용할 수 있는 아이템을 추가하고 아이템 생성기를 사용해 실시간으로 생성.

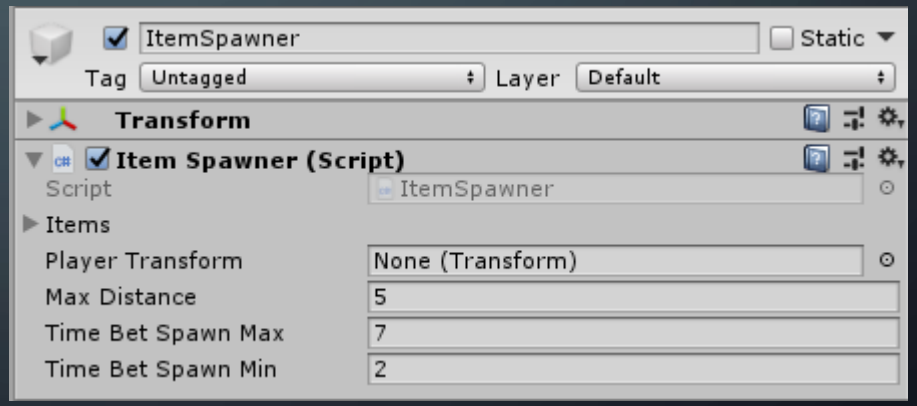
ItemSpawner Script의 기능

주기적으로 아이템 생성, 플레이어 근처의 **NaviMesh** 위에서 랜덤한 한 점을 선택하여 생성위치로 사용

ItemSpawner 추가

1. 빈 게임 오브젝트 생성(하이어라키 창에서 **Create > Create Empty),** 생성된 게임 오브젝트의 이름을 **Item Spawner** 로 변경
2. **Item Spawner** 게임 오브젝트에 **Script** 폴더의 **ItemSpawner Script** 추가





2. ITEM SPAWNER

ItemSpawner의 필드

```
public GameObject[] items; // 생성할 아이템들
public Transform playerTransform; // 플레이어의 트랜스폼
```

- **GameObject Type**의 배열 **item**에는 생성할 **Item Prefab**이 할당된다.
- **Transform Type**의 **playerTransform**에는 플레이어 캐릭터의 **Transform Component**가 할당되며 **player Trans form**은 플레이어의 위치를 파악하는데 사용한다.

```
public float maxDistance = 5f; // 플레이어 위치로부터 아이템이 배치될 최대 반경

public float timeBetSpawnMax = 7f; // 최대 시간 간격
public float timeBetSpawnMin = 2f; // 최소 시간 간격
private float timeBetSpawn; // 생성 간격

private float lastSpawnTime; // 마지막 생성 시점
```

- **timeBetSpawnMax**와 **timeBetSpawnMin**은 아이템 생성 시간 간격의 최대값과 최소값이다.
- **timeBetSpawn**은 다음 아이템 생성까지의 시간 간격이다. **timeBetSpawn**의 값은 **timeBetSpawnMin**과 **timeBet SpawnMax**사이의 값으로 결정된다.
- **lastSpawnTime**은 마지막으로 아이템을 생성한 시점.

Start() Method

- 생성 간격 **timeBetSpawn**을 최소값 **timeBetSpawnMin**과 최대값 **timeBetSpawnMax** 사이에서 랜덤결정, 마지막 생성 시점 **lastSpawnTime**을 **0**으로 초기화

```
private void Start()
{
    // 생성 간격과 마지막 생성 시점 초기화
    timeBetSpawn = Random.Range(timeBetSpawnMin, timeBetSpawnMax);
    lastSpawnTime = 0;
}
```

2. ITEM SPAWNER

- 생성된 적의 **onDeath** 이벤트에 등록한다.
  →**enemies.Remove(enemy) :** 자신을 리스트에서 제거
  →**Destroy(enemy.gameObject, 10f) : 10**초 뒤에 자신의 게임 오브젝트 파괴
  →**GameManager.instance.AddScore(100) :** 게임 점수를 **100**점 증가

```
// 적의 onDeath 이벤트에 익명 메서드 등록
// 사망한 적을 리스트에서 제거
enemy.onDeath += () => enemies.Remove(enemy);
// 사망한 적을 10초 뒤에 파괴
enemy.onDeath += () => Destroy(enemy.gameObject, 10f);
// 적 사망 시 점수 상승
enemy.onDeath += () => GameManager.instance.AddScore(100);
```

- 사망한 적을 적 리스트에서 빼고, 적 시체가 계속 늘어나지 않도록 사망한 적 게임 오브젝트를 **10**초 뒤에 파괴하고, 게임 점수를 **100**점 증가시킨다.
- 추가된 **Method**들은 **EnemySpawner Class**에 없다.
  →익명함수로 만들어져서 이벤트에 등록할 수 있다.

익명 함수와 람다식

- 미리 정의하지 않고, 실행 중인 코드 블록 내부에서 즉석 생성할 수 있는 **Method**
- 실시간으로 생성할 수 있으며, 변수에 저장할 수 있는 값이나 오브젝트로 취급되며, 생성된 익명 함수는 **Delegate Type** 의 변수에 저장할 수 있다.
  →미리 정의하지 않고 대부분 일회용으로 실시간 생성해서 사용하기 때문에 외부에서 따로 지칭할 수 있는 이름을 가지고 있지 않다.
- 간단한 메소드같은 경우에는 직접적인 함수 구현없이 실시간으로 생성해서 등록할 수 있다.

2. ITEM SPAWNER

Update() Method

매 프레임마다 현재 시점이 현재 시점이 아이템 생성을 처리하는 **Spawn()**을 실행할 수 있는 시점인지 체크하고**,** 가능한 경우 **Spawn() Method**를 실행한다**.**

```
// 주기적으로 아이템 생성 처리 실행
private void Update()
{
    // 현재 시점이 마지막 생성 시점에서 생성 주기 이상 지남
    // && 플레이어 캐릭터가 존재함
    if (Time.time >= lastSpawnTime + timeBetSpawn && playerTransform != null)
    {
        // 마지막 생성 시간 갱신
        lastSpawnTime = Time.time;
        // 생성 주기를 랜덤으로 변경
        timeBetSpawn = Random.Range(timeBetSpawnMin, timeBetSpawnMax);
        // 아이템 생성 실행
        Spawn();
    }
}
```

if문으로

1. **Time.time → lastSpawnTime + timeBetSpawn :** 마지막 생성 시점에서 생성 시간 간격 이상의 시간이 지났나**?**
2. **playerTransform != null : (**생성 위치의 기준이 될**)** 플레이어의 **Transform Component**가 존재하는가**?**

를 체크하고 있다**.**

조건을 만족하면 **Spawn()**을 실행하면서 마지막 시간과 생성주기를 재설정한다**.**

2. ITEM SPAWNER

Spawn() Method

- 플레이어의 위치에서 일정 반경 내부의 **NaviMesh** 위의 랜덤한 위치를 찾아 그곳에 아이템을 생성

```
// 실제 아이템 생성 처리
private void Spawn()
{
    // 플레이어 근처에서 내비메시 위의 랜덤 위치 가져오기
    Vector3 spawnPosition =
        GetRandomPointOnNavMesh(playerTransform.position, maxDistance);
    // 바닥에서 0.5만큼 위로 올리기
    spawnPosition += Vector3.up * 0.5f;

    // 아이템 중 하나를 무작위로 골라 랜덤 위치에 생성
    GameObject selectedItem = items[Random.Range(0, items.Length)];
    GameObject item = Instantiate(selectedItem, spawnPosition, Quaternion.identity);

    // 생성된 아이템을 5초 뒤에 파괴
    Destroy(item, 5f);
}
```

- 플레이어의 위치를 나타내는 **playerTransform**을 중심으로 **maxDistance** 반경내부에서 **NaviMesh** 위의 랜덤 위치를 찾는다**.**
- **GetRandomPointOnNavMesh**를 통해 **maxDistance**에 랜덤위치를 받아서 사용하고**, spawn**위치를 바닥에 아이템 이 딱 붙어서 생성되지 않도록 **spawnPosition**의 **y**값을 **0.5**만큼 높인다**.**
- **Item**에 할당된 여러 **ItemPrefab** 중 생성할 **ItemPrefab**을 하나 랜덤 선택하고 **Instantiate() Method**로 **Item Pref ab**의 복제본을 생성한다**,** 복제본을 배치할 위치는 **spwanPosition,** 회전은 **Quaternion.identity(**오일러각**(0, 0, 0)** 회전에 대응**)**이다**.**
- 생성된 **Item**이 **5**초 동안 아무 일도 벌어지지 않으면 자동적으로 삭제된다**.**

2. ITEM SPAWNER

GetRandomPointOnNavMesh() Method

- 아이템 생성기는 **NavMesh** 위의 랜덤한 위치를 선택해 아이템을 생성해야 한다.
  →입력된 **center** 를 중심으로 **distance** 반경 안에서 **NavMesh** 위의 랜덤한 한 점을 찾아서 반환

```
// 내비메시 위의 랜덤한 위치를 반환하는 메서드
// center를 중심으로 distance 반경 안에서 랜덤한 위치를 찾는다
private Vector3 GetRandomPointOnNavMesh(Vector3 center, float distance)
{
    // center를 중심으로 반지름이 maxDistance인 구 안에서의 랜덤한 위치 하나를 저장
    // Random.insideUnitSphere는 반지름이 1인 구 안에서의 랜덤한 한 점을 반환하는 프로퍼티
    Vector3 randomPos = Random.insideUnitSphere * distance + center;

    // 내비메시 샘플링의 결과 정보를 저장하는 변수
    NavMeshHit hit;

    // maxDistance 반경 안에서, randomPos에 가장 가까운 내비메시 위의 한 점을 찾음
    NavMesh.SamplePosition(randomPos, out hit, distance, NavMesh.AllAreas);

    // 찾은 점 반환
    return hit.position;
}
```

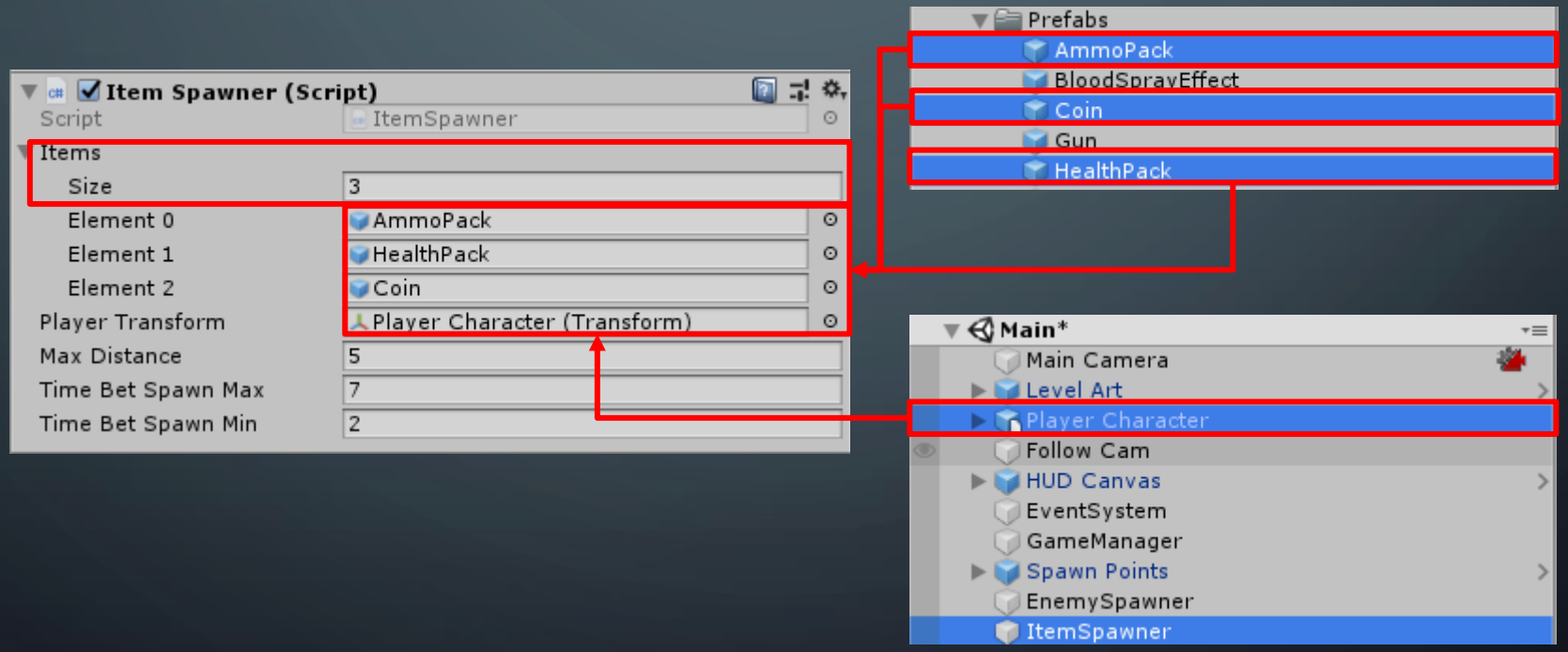
- **Center** 를 중심으로 **distance** 만큼의 반지름을 가지는 구가 있다고 가정하고, 해당 구의 내부에서 램덤한 한 점을 선택, 그리고 해당 점의 위치를 **randomPos**에 할당한다.
- **Random.insideUnitSphere** 프로퍼티는 반지름이 **1**유닛인 구 내부의 한 점을 반환한다, 여기에 **distance**를 곱하고 **center**를 더하면 위치가 **cente**이며 반지름이 **distanc**인 구 내부의 랜덤한 한 점을 선택하는 것과 같고 그다음 **NaviM esh** 샘플링을 실행하여 **randomPos**와 가장 가까운 **NavMesh** 위의 한 점을 찾는다.
  →**NavMesh** 샘플링은 특정 반경 내부에서 어떤 위치와 가장 가까운 **NavMesh** 위의 한 점을 찾는 처리
- **NavMesh** 샘플링의 실행 결과는 **Raycast**처럼 별개의 정보 저장용 변수에 할당된다. 그렇기 때문에 **NavMesh**샘플링 정보를 저장할 **NavMeshHit Type**의 변수를 선언한다.

그렇게 생성된 포지션을 반환한다.

2. ITEM SPAWNER

ItemSpawner Component 설정

1. **ItemSpawner Component**의 **Items**필드에서 **Size**를 **3**으로 변경
2. **Prefabs**폴더의 **AmmoPack, Coin, HealthPack** 프리팹을 **Items**의 각 원소에 할당
3. **Player GameObject**를 **PlayerTransform Field**로 **Drag&Drop**하여 할당

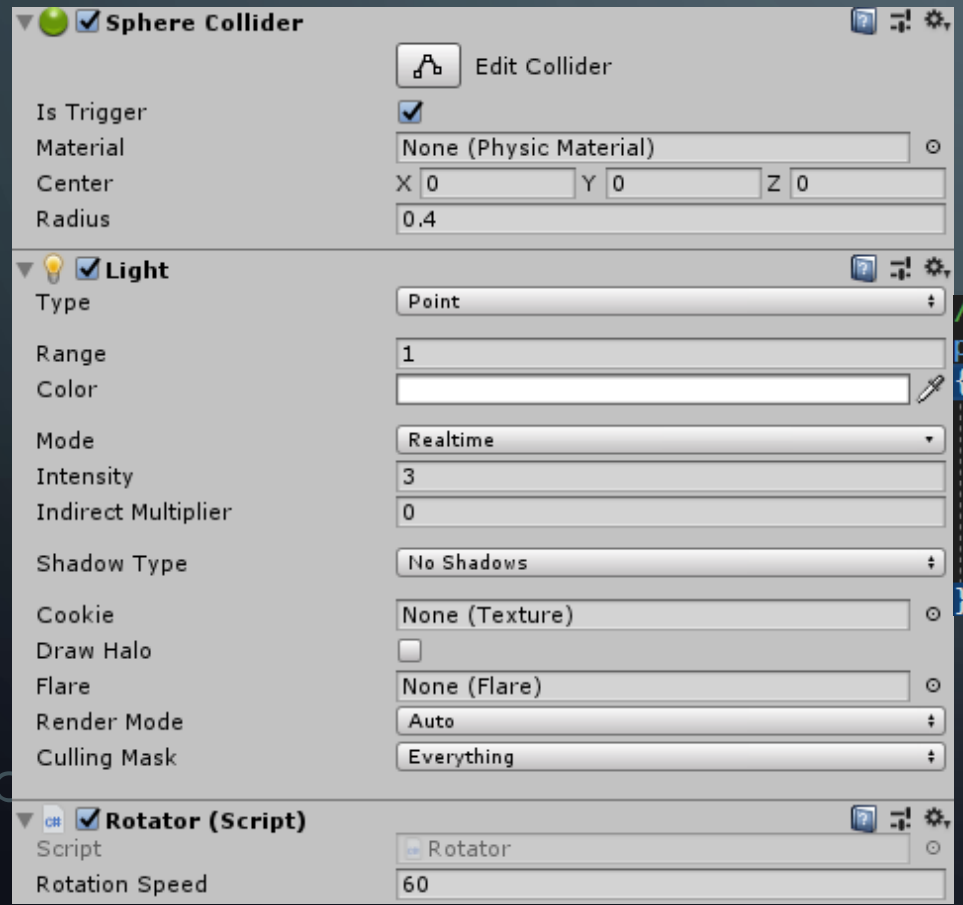


3. ITEMPREFAB

AmmoPack, Coin, HealthPack ItemPrefab은 각각 남은 탄알을 증가시키고**,** 게임 점수를 증가시키며**,** 플레이어 캐릭터 체력을 증가시키는 부분이 필요하다**.**
ItemPrefab들은 공통적으로 **Trigger** 설정된 **Sphere Collider, Light, Rotator(Prefab**을 회전시키는**) Script 3**가지의 **Component**를 가지고 있다**.**

Rotator Script

실시간으로 회전하는 스크립트




```
// 게임 오브젝트를 지속적으로 회전하는 스크립트
public class Rotator : MonoBehaviour
{
    public float rotationSpeed = 60f;

    private void Update() {
        transform.Rotate(0f, rotationSpeed * Time.deltaTime, 0f);
    }
}
```

3. ITEMPREFAB

Coin Script

GameManager에 접근해서 **AddScore() Method**를 실행하여 점수를 추가한다.
사용된 아이템은 사라져야 하므로 **Use() Method** 마지막에 **Destroy() Method**를 실행한다.

```
public class Coin : MonoBehaviour, IItem
{
    public int score = 200; // 증가할 점수

    public void Use(GameObject target)
    {
        // 게임 매니저로 접근해 점수 추가
        GameManager.instance.AddScore(score);
        // 사용되었으므로, 자신을 파괴
        Destroy(gameObject);
    }
}
```

AmmoPack Script

입력된 **GameObject**의 **PlayerShooter Component**에 접근한다.
PlayerShooter Component를 통해 **Player**가 사용 중인 총에 접근하고, 총의 남은 탄알을 증가시킨다.

```
public class AmmoPack : MonoBehaviour, IItem
{
    public int ammo = 30; // 충전할 총알 수

    public void Use(GameObject target) {
        // 전달 받은 게임 오브젝트로부터 PlayerShooter 컴포넌트를 가져오기 시도
        PlayerShooter playerShooter = target.GetComponent<PlayerShooter>();

        // PlayerShooter 컴포넌트가 있으며, 총 오브젝트가 존재하면
        if (playerShooter != null && playerShooter.gun != null)
        {
            // 총의 남은 탄환 수를 ammo 만큼 더한다
            playerShooter.gun.ammoRemain += ammo;
        }

        // 사용되었으므로, 자신을 파괴
        Destroy(gameObject);
    }
}
```

3. ITEMPREFAB

HealthPack Script

입력 받은 상대방 **GameObject**로부터 **LivingEntity Type**의 **Component**를 찾아 **RestoreHealth() Method**를 실행하여 체력을 증가시킨다.

```
public class HealthPack : MonoBehaviour, IItem
{
    public float health = 50; // 체력을 회복할 수치

    public void Use(GameObject target) {
        // 전달받은 게임 오브젝트로부터 LivingEntity 컴포넌트 가져오기 시도
        LivingEntity life = target.GetComponent<LivingEntity>();

        // LivingEntity컴포넌트가 있다면
        if (life != null)
        {
            // 체력 회복 실행
            life.RestoreHealth(health);
        }

        // 사용되었으므로, 자신을 파괴
        Destroy(gameObject);
    }
}
```

POST PROCESSING

1. POST PROCESSING STACK

포스트 프로세싱(Post Processing)

- 게임의 영상미를 추가하는 방법.
- 후처리라고 부르며, 게임 화면이 최종 출력되기 전에 카메라의 이미지 버퍼에 삽입하는 추가 처리이다.
- 적은 노력으로 뛰어난 영상미를 구현할 수 있다.

- 카메라 앱의 필터링 같은 것과 비슷하다고 이해하자!
- 연산은 랜더링 파이프라인의 주요 과정에 적용되지 않고 마지막 부분에 적용된다.
- **Unity**는 포스트 프로세싱을 쉽게 사용할 수 있는 포스트 프로세싱 스택 패키지를 제공한다.
  →프로젝트 생성시 사용여부에 따라 기본 포함 프로젝트가 있는 이유
  →**V1**과 **V2** 버전이 존재한다.

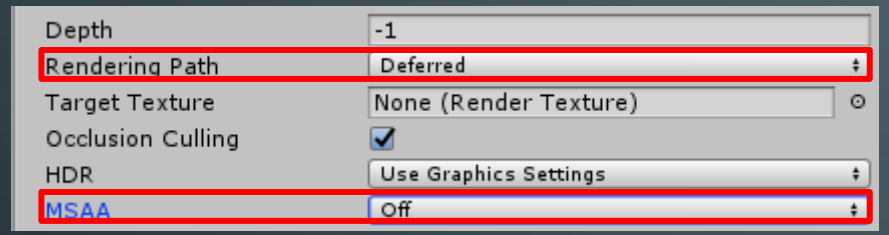
2. POST PROCESSINGT 설정

렌더링 경로 설정

- 포스트 프로세싱을 적용하기 전에 최적의 품질을 얻기 위해 카메라 렌더 설정을 변경한다.

카메라 렌더 설정 변경

1. 하이어라키 창에서 **Main Camera GameObject** 선택
2. **Camera Component**의 **Rendering Path**를 **Deferred**로 변경
3. **Allow MSAA**를 해제**(off)**



- **Camera Component**의 렌더링 경로는 렌더링이 처리되는 순서와 방법을 결정하는 옵션이다. 기본값인 **Use Graphics Settings**는 프 로젝트 설정에 맞춰 자동으로 렌더링 경로를 결정하며, 일반적으로 포 워드 랜더링**(Forward Rendering)**옵션을 설정한다.

- 포워드 렌더링은 성능이 가볍지만 라이팅 표현이 실제보다 간략화되고 왜곡되어 있다, 이것을 디퍼드 세이딩**(Deferred Sheading)**으로 바꿔 온전하게 표현해주자.

포워드 렌더링

- 각각의 오브젝트를 그릴 때마다 해당 오브젝트에 영향을 주는 모든 라이팅도 함께 계산한다.
- 메모리 사용량이 적고 저사양에서도 비교적 잘 동작한다. 하지만 연산 속도가 느리며, 오브젝트와 광원이 움직이거나 수가 많아질수록 연산량이 급증하여 사용하기 힘들다.
- 하나의 게임오브젝트에 대해 최대 **4**개의 광원만 제대로 개별 연산한다, 나머지 **'**중요하지 않은**'** 광원과 라이팅 효과는 부하를 줄이기 위해 합쳐서 한 번에 연산한다. 따라서 라이팅 효과가 실제와 다르게 표현될 수 있다.

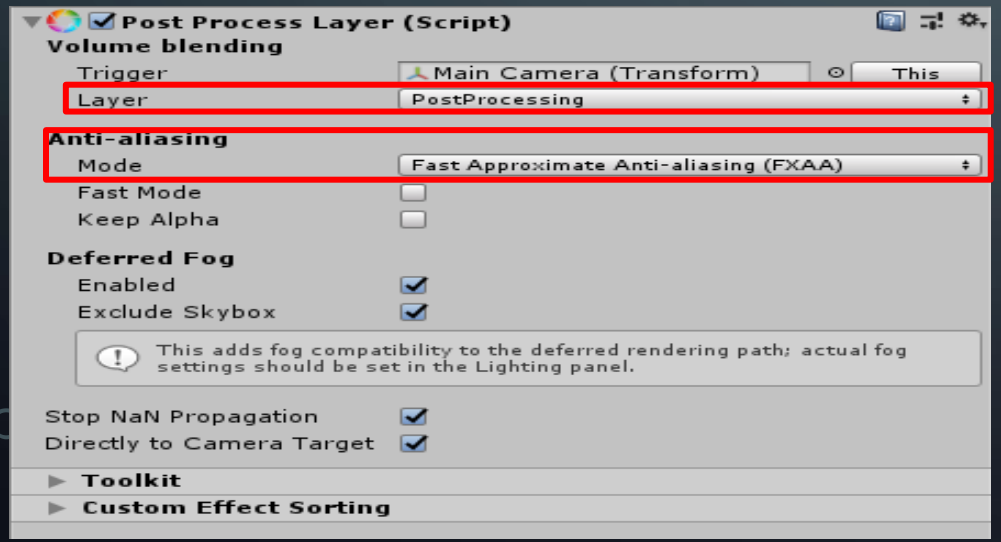
2. POST PROCESSINGT 설정

디퍼드 셰이딩

- 라이팅 연산을 미뤄서 실행하는 방식
- 첫 번째 패스에서는 오브젝트의 메시를 그리되 라이팅을 계산하거나 색을 채우지 않는다. 대신 오브젝트의 여러 정보를 종류별로 버퍼에 저장한다.
- 두 번째 패스에서 첫 번째 패스의 정보를 활용해 라이팅을 계산하고 최종 컬러를 결정한다.
- **Unity**의 디퍼드 셰이딩은 개수 제한 없이 광원을 표현할 수 있다. 또한 모든 광원의 효과가 올바르게 표현된다.
- 단, 디퍼드 셰이딩은 **MSAA** 같은 일부 안티엘리어싱(계단 현상 제거) 설정을 제대로 지원하지 않는다. 그래서 카메라 **Component**의 **MSAA** 설정을 체크 해제해줘야 한다.

카메라에 포스트 프로세싱 적용

1. **Main Camera GameObject**에 **Post-process Layer Component** 추가(**Add Component > Rendering > Pos t-process Layer**)
2. **Post-process Layer Component**의 **Layer**를 **PostProcessing**으로 변경
3. **Anti-aliasing**의 **Mode**를 **FXAA**로 변경



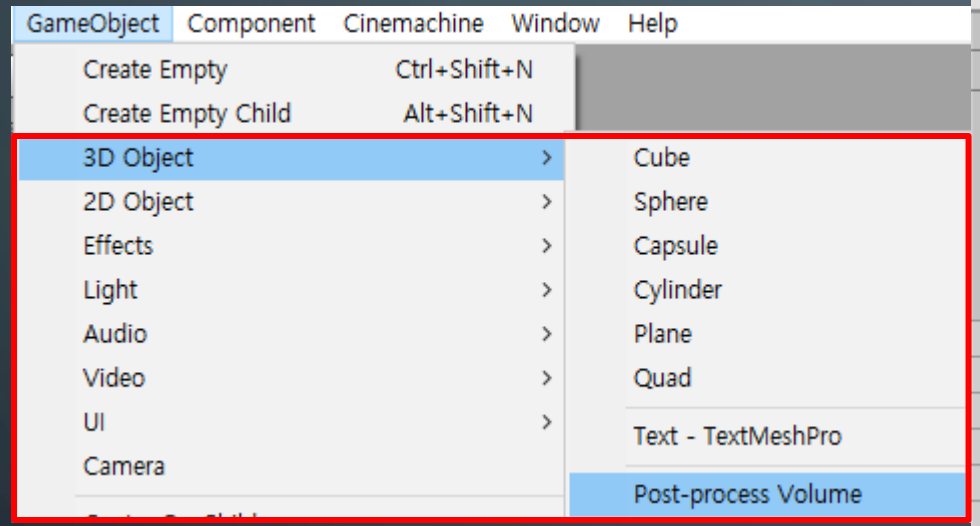
포스트 프로세스 레이서는 포스트 프로세싱 볼륨을 감지하고 포스트 프로세싱 볼륨으로부터 설정을 얻어와 카메라에 적용. 단, **Scene**의 모든 **GameObject**에 대해 포스트 프로세싱 볼륨을 찾으려 하면 성능에 악영향을 미친다. 그렇기 때문에 포스트 프로세스 레이어는 특정 레이어에 대해서만 포스트 프로세싱 볼륨을 감지하도록 설정해야 한다.

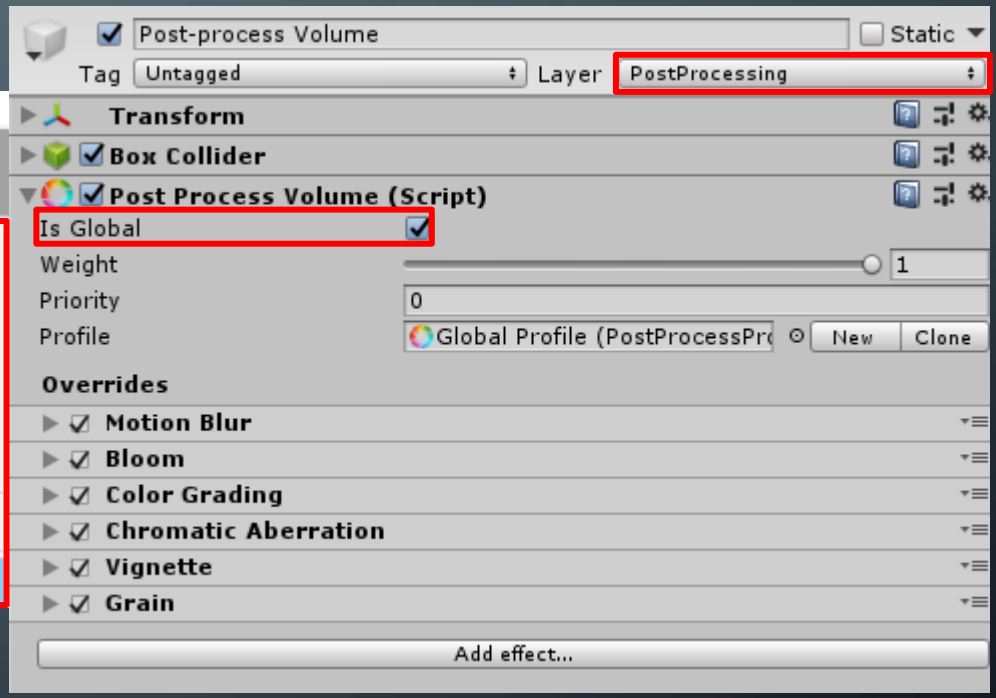
Layer 설정을 통해 **Postprocessing**이라는 레이어를 가진 게임오브젝트만 감지하도록 한것이다.
안티앨리어싱은 비교적 부하가 적은 **FXAA**로 설정.

2. POST PROCESSINGT 설정

포스트 프로세스 볼륨 추가

1. **Post-process Volume GameObject** 생성**(Create > 3D Object > Post-process Volume)**
2. **Post-process Volume GameObject**의 **Layer**를 **PostProcessing**으로 변경**, Is Global**를 체크



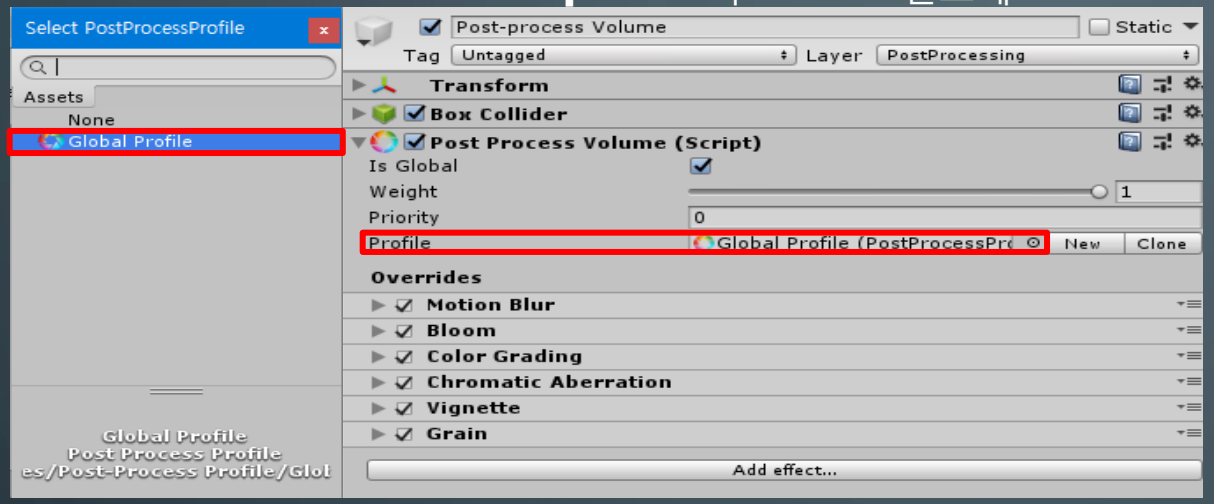


- 본래 트리거 콜라이더와 함께 사용해야한다 포스트 프로세스 볼륨의 콜라이더와 포스트 프로세스 레이어의 **Trigger** 필드 에 할당된 게임 오브젝트의 위치가 겹치면 해당 포스트 프로세스 레이어의 **Trigger** 필드에 할당된 게임 오브젝트를 거쳐 카메라에 적용된다**.**
- 현재 카메라의 위치가 어디든 일괄적으로 효과를 적용하면 되는 상황이기때문에 포스트 프로세스 볼륨 컴포넌트의 **Is Global**을 체크하여 위치와 상관없이 효과를 전역으로 사용한다**.**

2. POST PROCESSINGT 설정

포스트 프로세스 프로파일 할당

Post Process Volume Component의 **Profile** 필드에 **Global Profile** 할당



Global Profile은 포스트 프로세스 설정파일.

사용할 효과 목록을 기록하는 프리셋 파일, **Profile** 필드 옆의 **New** 버튼을 클릭해 새로운 프로파일을 생성하거나, 누군가가 기존에 만들어둔 프로파일을 가져와서 사용할 수 있다.

사용된 효과들

모션 블로(Motion Blur) : 빠르게 움직이는 물체에 대한 잔상효과

블룸(Bloom) : 뽀샤시효과, 밝은 물체의 경계에서 빛이 산란되는 효과

컬러 그레이딩(Color Grading) : 사진필터(따뜻한, 차가운등등) 효과, 최종 컬러, 대비, 감마 등을 교정

색 수차(Chromatic Aberration) : 방사능 효과, 이미지의 경계가 번지고 삼원색이 분리되는 효과, 게임에서 방사능이나 독 중독 효과를 표현할 때 주로 사용

비네트(Vignette) : 화면 가장자리의 채도와 명도를 낮추는 효과, 화면 중심에 포커스를 주고 차분한 느낌을 줄 때 사용

그레인(Grain) : 화면에 입자 노이즈 추가, 필름 영화 같은 효과를 내거나, 공포 분위기를 강화할 때 사용.