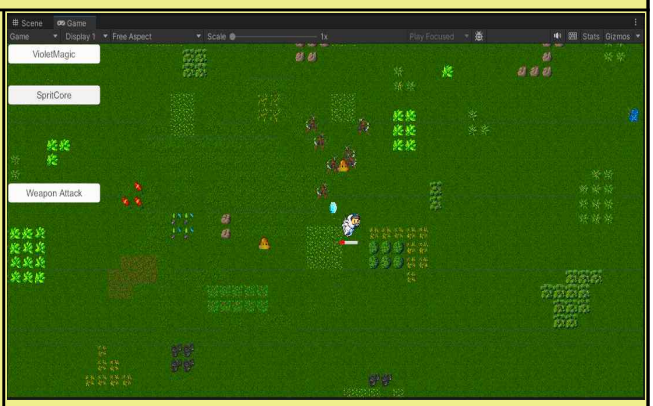
[컴퓨터그래픽스II 계획서 양식]

프로젝트 계획서

주제명	핵 앤 슬래시 장르의 2D 게임 만들기	
작성자	성명	E-mail
	백무송	musong0611@naver.com

〈 목표 정의 〉

최종 구현의 예(참조 또는 손그림)



목표 정의

- 무작위 몬스터를 출몰(생성)하는 방법
- 무작위 몬스터가 플레이어를 따라가는 기능 구현
- 메인 캐릭터(플레이어)를 조종할 수 있는 기능 구현
- 무작위 몬스터와 메인 캐릭터(플레이어)가 만날 시 충돌처리 과정 구현
- 메인 캐릭터(플레이어) 체력 바 생성
- 유니티 애니메이터를 통해 메인 캐릭터(플레이어)와 몬스터 이동의 자연스러움 구현
- **몬스터의 공격모션 구현(미완성-구현 시 오류 발생)**
- 게임 캐릭터들의 활동 부분인 맵(필드) 제작하기
- 버튼 기능을 통한 무기를 바꾸는 기능 추가

〈 목표 정의 1 〉

정의명	핵 앤 슬래시 장르의 2D 게임 만들기를 위한 기본 설계도
접근 방법	● 캐릭터 모델링(몬스터,플레이어)은 에셋스토어에 받아서 사용합니다. (밑에 참고한 에셋 링크 첨부) ● 메인 캐릭터(플레이어) 시작 위치는 맵 중앙으로 고정하고 플레이어 주위에 일정 범위의 원의 영역을 생성해 따라가게 한다. (고민중인 내용) ● 몬스터 생성 방법은 메모리 풀링을 이용하고 list를 이용해 데이터가 꽉 차면 list를 다시 할당하여 5개를 메모리를 할당하는 식으로 만듭니다. (고민 중인 내용) ● 몬스터 생성 위치는 플레이어에 설정한 원 범위 주위에서 생성되게 설정합니다. ● 몬스터가 캐릭터를 따라가게 하는 방법은 moveTowards를 이용하여 캐릭터를 따라가게 만들고, 몬스터가 이동하는 방향의 x축이 캐릭터 위치에 있는 방향의 반대를 향하고 있을 때 x축 반대에 위치 할 때 회전을 사용하지 않고 Flip 옵션을 이용해 방향을 바꿔 다시 다가오게 만듭니다. ● 플레이어에 physics2d.overlapcircle 또는 colider를 이용해 physics2d.overlapcircle 경우 거리에 따라 먼저 충돌했는지에 따라 충돌 판정을 해주고, colider인 경우 그냥 일정 범위안에 있을 시 충돌 판정을 하게한다. (colider만 사용시 동시에 몬스터가 범위안에 들어올 경우 오류가 발생할 수도 있다.) ● 유니티 기능중 Animator기능을 이용해 에셋에 있는 여러 가지 모션을 조건을 통해 연결해 몬스터와 플레이어의 이동시 자연스러운 모션을 구현합니다. ● 맵은 2D tilemap을 이용해 맵을 생성한다. (사용 예정인 타일 링크는 아래 첨부) ● 유니티의 기능 Animator기능을 이용해 몬스터 공격모션을 구현하고 조건은 플레이어의 physics2d.overlapcircle 또는 colider의 범위안에 있을 시 공격모션이 나가도록 한다. ● 무기 변경 기능은 UI의 Legacy Button을 이용해 버튼으로 만들어 클릭 시 무기 모습이 바뀌는 식으로 설정합니다. ● 플레이어 체력 바는 UI의 Slider기능을 이용해 체력을 설정하고 색깔은 빨간색으로 지정합니다.

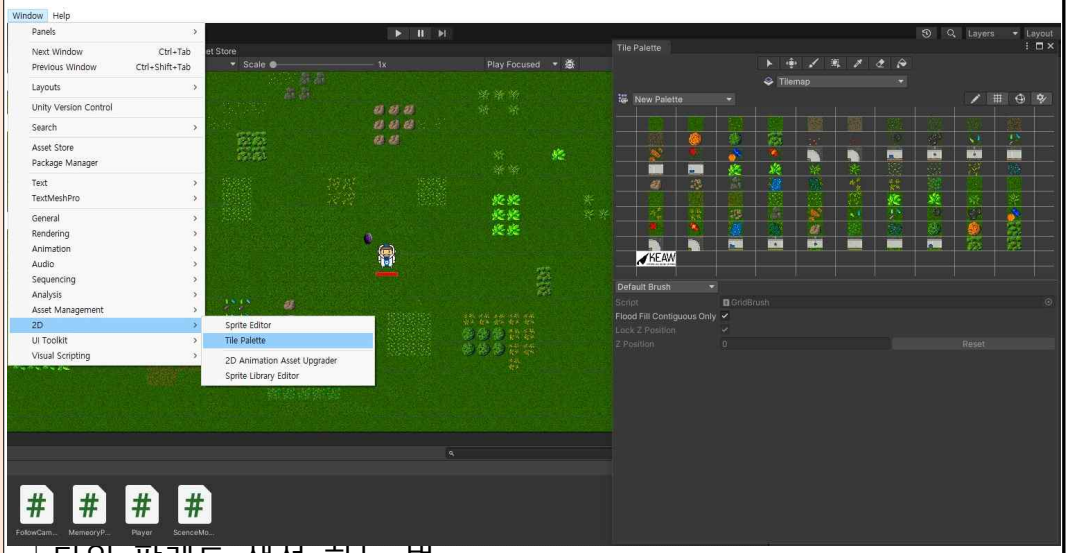
(지면 부족시 추가하여 작성)

결과 및 분석

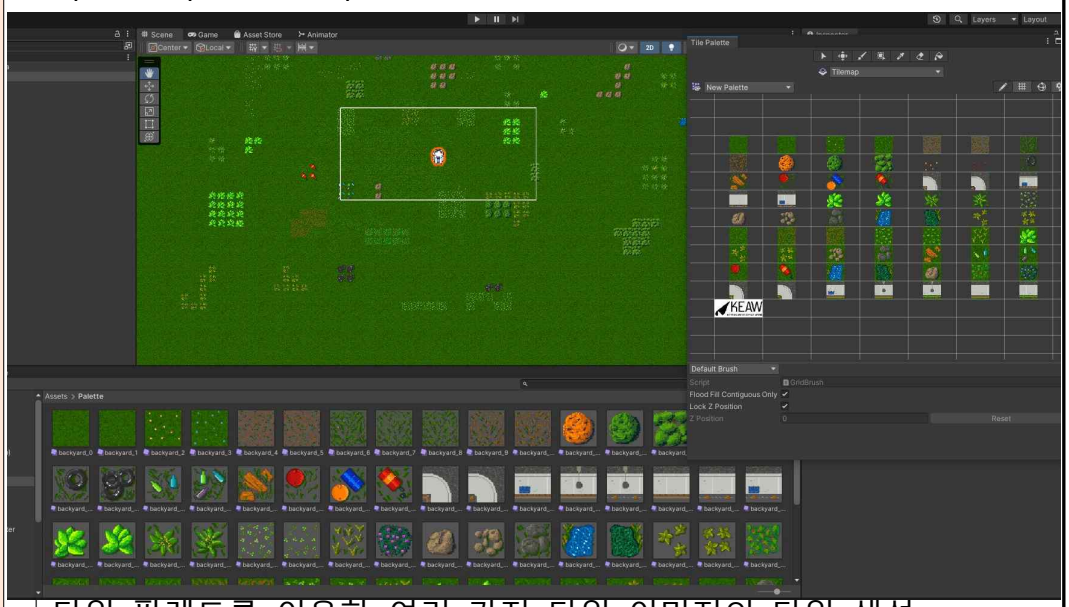
● **게임 캐릭터들의 활동 부분인 맵(필드) 제작하기**

Tilemap을 사용해 맵을 제작합니다.
-Tilemap 컴포넌트란-> 2D 레벨을 생성하기 위해 타일 에셋을 저장하고 처리하는 시스템입니다.
-Tile Palette 창에서 선택한 타일로 팔레트를 만듭니다. 이러한 타일은 타일맵에서 페인팅하는 데 사용됩니다.

*제작 과정
1.사각형 틀의 타일 맵을 생성한다.
2.윈도우 창에 2D의 타일 팔레트를 생성한다.
2-1 각 사각형 틀 하나하나에 그림을 새겨 넣는 방식
2-2 Create New Palette를 클릭해 팔레트 폴더를 만들고 에셋스토어 다운받은 맵에 Slied 파일을 넣는다.
2-3 각 칸에 자기가 넣고 싶은 그림을 범위를 선택해 틀에 맞게 필드에 채워 넣습니다.



↳타일 팔레트 생성 하는 법



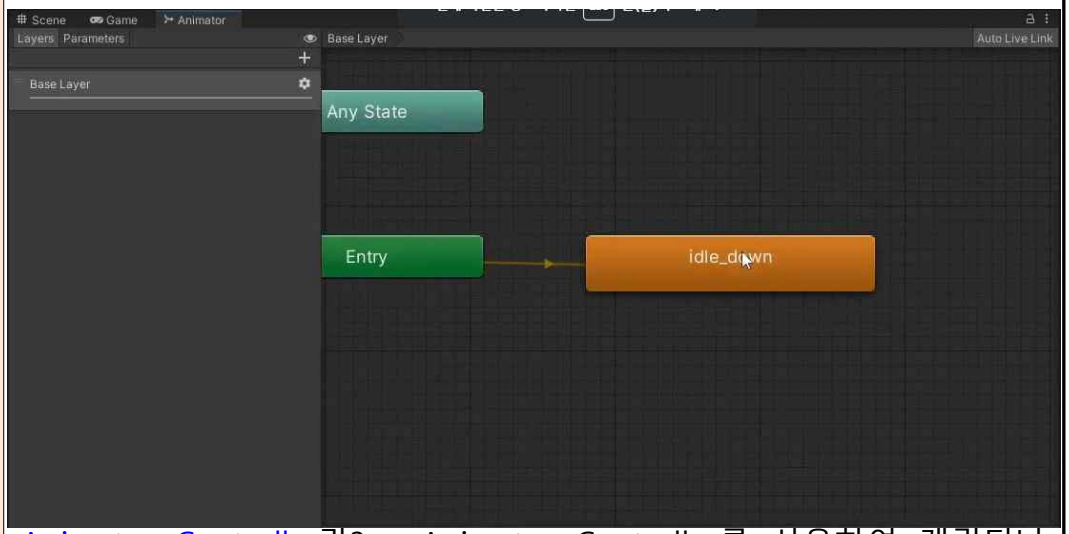
↳타일 팔레트를 이용한 여러 가지 타일 이미지의 타일 생성

결과 및 분석

- **메인 캐릭터(플레이어)를 조종할 수 있는 기능 구현**
- **유니티 애니메이터를 통해 메인 캐릭터(플레이어)와 몬스터 이동의 자연스러움 구현**

(1) animator을 이용한 오브젝트 움직임 구현

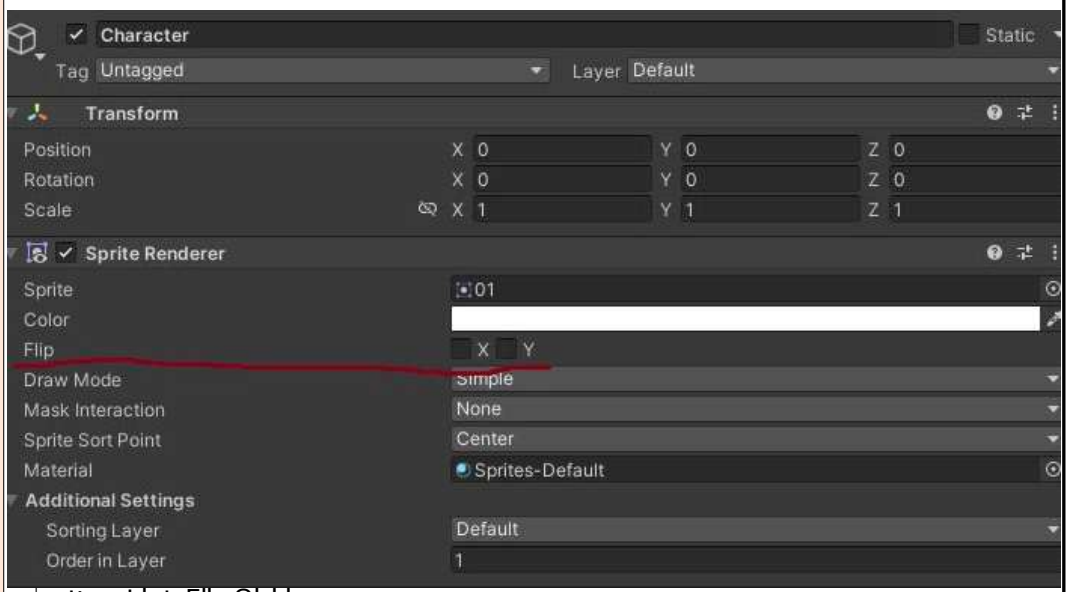
1. 에셋 스토어에 캐릭터를 이용할 에셋을 다운받습니다.
2. 다운 받은 캐릭터를 animator를 이용해 움직임을 표현 할껍니다.
3. animator란? -> 애니메이션을 효율적으로 관리해주는 역할입니다.
4. 다운 받은 곳에 있는 idle파일이름에 존재하는 01이라는 sprite형식의 파일을 GameScene에 넣고 다운받은 Animator Controller를 넣어 멈춰 있는 상태의 애니메이션을 구성합니다.



Animator Controller란?-> Animator Controller를 사용하여 캐릭터나 오브젝트의 애니메이션 클립 세트와 관련 애니메이션 전환을 정렬하고 관리할 수 있습니다.

(2) 좌우로 움직일 시 방향 전환

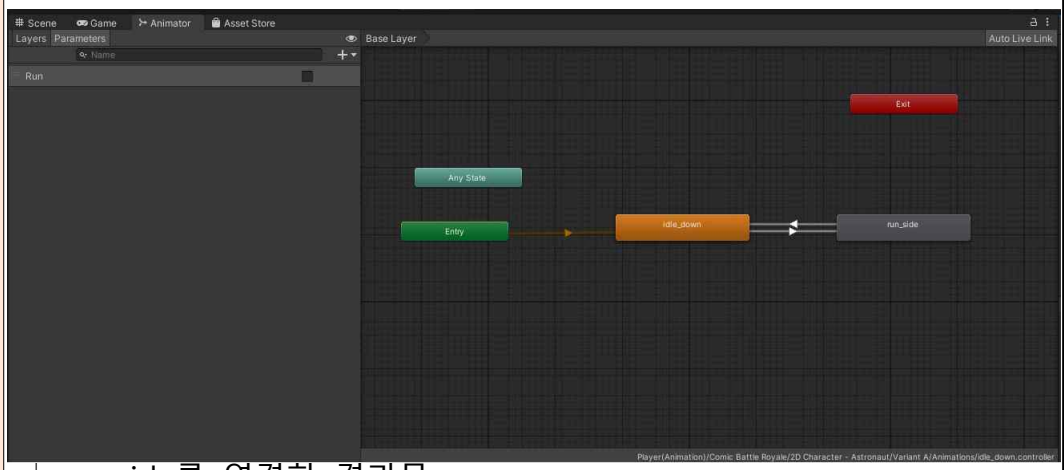
Sprite Renderer에서 Flip X와 Y를 체크하면 좌우, 상하로 이미지를 반전시킬 수 있다.



↳Flip 시스템 위치

결과 및 분석

캐릭터의 애니메이터에 run side 애니메이션을 애니메이터 컨트롤러를 넣고 idle_down과 연결한다.



↳run side를 연결한 결과물

Flip방향 전환 하는 핵심 코드

```
if (direction.x > 0 )
//0보다 x의 direction이 크다 (오른쪽으로 움직인다. 즉 애니메이션의 움직임은 왼쪽이기 때문에 filpX를 체크해 방향 전환을 시킨다)
{
    spriteRenderer.flipX = true;
}
else if (direction.x < 0 )//위의 상황과 반대인 경우
{
    spriteRenderer.flipX = false;
}
```

코드 해석

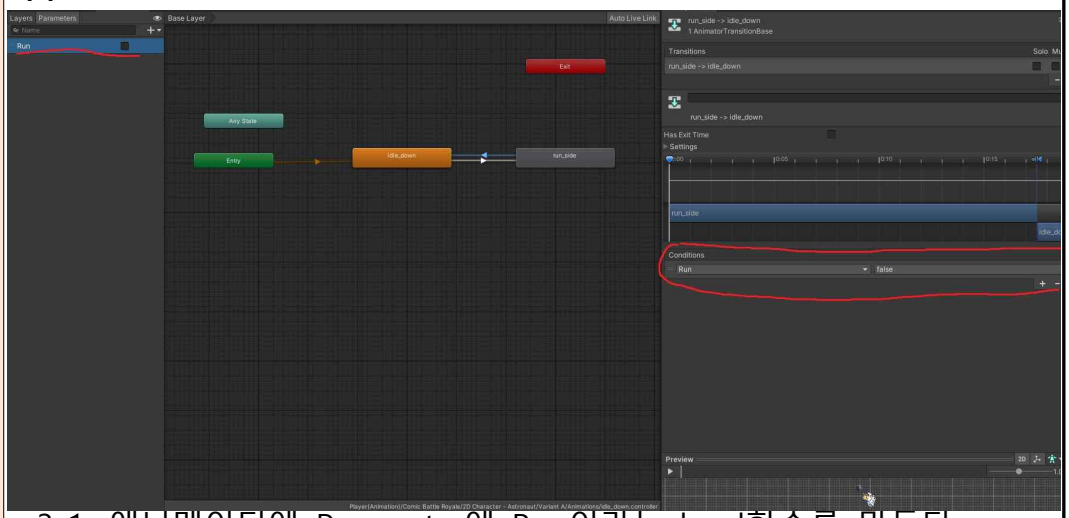
이 코드가 실행되면, 'direction.x'의 값에 따라 스프라이트의 방향을 조절합니다.

'direction.x'의 값이 0보다 크면, 즉 오른쪽으로 움직이는 경우, 'spriteRenderer.flipX'는 'true'로 설정됩니다. 이는 스프라이트 이미지를 x축 기준으로 뒤집어, 왼쪽을 보도록 만드는 것입니다.

반대로 'direction.x'의 값이 0보다 작으면, 즉 왼쪽으로 움직이는 경우, 'spriteRenderer.flipX'는 'false'로 설정됩니다. 이는 스프라이트 이미지를 원래대로 두어, 오른쪽을 보도록 만드는 것입니다.

결과 및 분석

(3) run side 애니메이션을 실행시키기위한 조건 설정

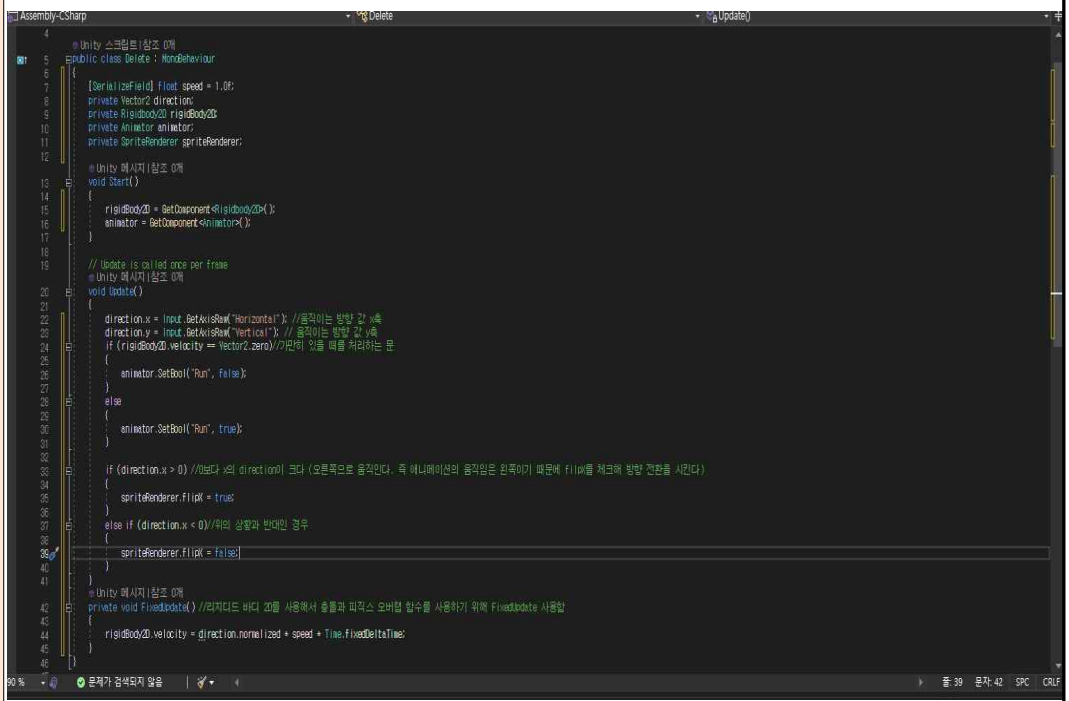


3-1. 애니메이터에 Parameter에 Run이라는 bool함수를 만든다

3-2. idle_down에서 run side로 가는 애니메이션은 Conditons에 Run 함수를 추가해 (true)로 설정

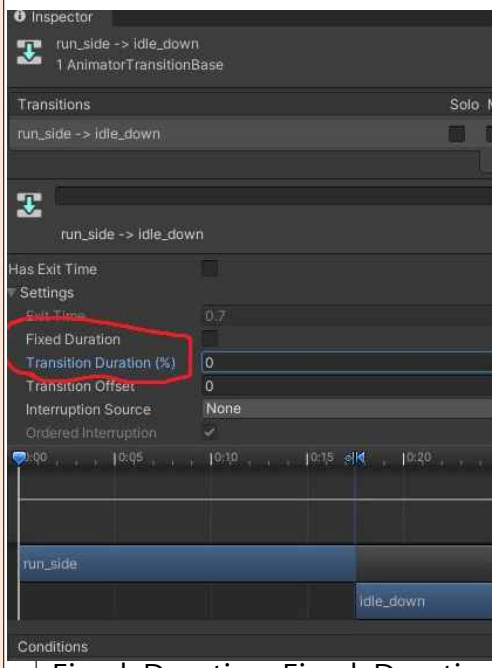
run side에서 idle_down로 가는 애니메이션은 Conditons에 Run 함수를 추가해 (falase)로 설정

(4) 플레이어를 조종하는 코드 + 움직일때 방향 전환을 하는 코드

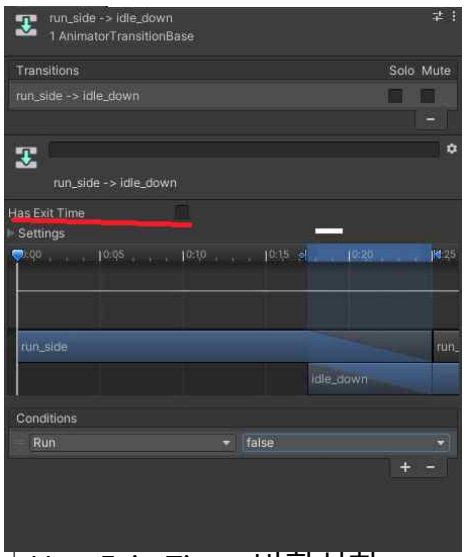


결과 및 분석	**위 그림의 플레이어를 조종하는 코드 + 움직일때 방향 전환을 하는 코드 해석** Update() 함수: 이 함수는 프레임마다 호출되며, 이 곳에서 캐릭터의 움직임과 애니메이션을 제어합니다. Input.GetAxisRaw("Horizontal")과 Input.GetAxisRaw("Vertical")을 통해 x축과 y축 방향의 입력을 받아 direction 변수에 저장합니다. 캐릭터가 움직이지 않을 때는 "Run" 애니메이션을 멈추고, 움직일 때는 "Run" 애니메이션을 실행합니다. FixedUpdate() 함수: 이 함수는 물리 연산을 처리할 때 호출되며, 여기서는 Rigidbody2D의 속도를 조절하여 캐릭터를 움직입니다. direction.normalized는 방향 벡터를 정규화(크기를 1로 만드는 과정)하여, 속도가 일정하게 유지되도록 합니다. speed * Time.fixedDeltaTime는 프레임 간격을 고려하여 캐릭터의 움직임을 부드럽게 만듭니다. **(5)**움직임 부자연스러움 해결하는 법 Has Exit Time을 활성화 할 경우 해당 동작을 끝내고 다음 동작으로 이동하게 되고 비활성화 할 경우 conditions에 조건이 충족될 때 즉시 동작을 전환한다. 그러므로 자연스러운 움직임을 위해 비활성화를 시킨다. Fixed Duration-Fixed Duration 상자에 체크되어 있을 경우 전환 시간은 초로 나타납니다. Fixed Duration 상자가 체크되어 있지 않을 경우 전환 시간은 소스 상태의 정규화된 시간의 부분으로 나타납니다. 그러므로 Fixed Duration를 비활성화 시키고 값을 0으로 해서 즉각 반응하게 만든다.

결과 및 분석



↳Fixed Duration-Fixed Duration 비활성화



↳Has Exit Time 비활성화

결과 및 분석

● **무작위 몬스터를 출몰(생성)하는 방법**

1. 몬스터를 생성하는 빈 오브젝트(Create Empty)를 Scene에 추가하고 CreatUnit이라는 스크립트 생성합니다.

*저는 몬스터를 관리하는데 있어서 메모리 풀을 사용합니다.

메모리 풀이란?

유니티에서 Instantiate와 Destroy는 상당히 무거운 함수입니다.
결국에 미리 프리팹을 만들어놓지 않으면 Instantiate는 어쩔 수 없이 동반해야하는 작업일 것이지만, Destroy는 우리가 약간의 트릭을 사용해서 부수지 않고, 이후의 Instantiate를 사용할 자리에 부수지 않은 오브젝트를 재사용하면 지속적으로 성능향상이 이루어질 것입니다.
저는 메모리 풀이라는 배열과 같은 형태의 자료구조에 오브젝트들을 미리 담아놓고, Destroy를 요청한다면 없애는 것이 아니라 SetActive(false)를 이용해 단순히 비활성화만 시킨 후에, 이후 Instantiate를 요청한다면 SetActive(true)로 바꿔주고 해당 오브젝트를 다시 돌려주는 기능으로 구성할 것입니다.

2.MemeoryPool 스크립트 생성

결과

및

분석

MemeoryPool 스크립트 코드(해석 나오기 전까지 이어진 코드입니다.)

```
public class MemeoryPool
{
        public class PoolItem
        {
            public GameObject gameObject;
// 실제 화면에 나타나는 게임 오브젝트
        }
        private int increaseCount =1;
//게임 오브젝트가 부족할 때 추가시키는 개수
        private int MaxCount; //현재 리스트에 등록되어 있는 게임 오브젝트 개수
        private int activeCount; //현재 활성화되어 있는 게임 오브젝트 개수
        private GameObject poolObject;
//오브젝트 풀링에서 관리하는 게임 오브젝트
        private List<PoolItem> poolItemList;
//모든 오브젝트를 관리하고 저장하는 리스트
    public MemeoryPool(GameObject poolObject)
    {
        MaxCount =0;
        activeCount =0;
        this.poolObject = poolObject;
        poolItemList =new List<PoolItem>();
        InstantiateObject(this.poolObject);
    }
    public void InstantiateObject(GameObject poolObject)
    {

        MaxCount += increaseCount; //값을 넣어준다는 뜻
        for (int i =0; i < increaseCount; i++)
        {
            PoolItem poolItem =new PoolItem();
            this.poolObject = poolObject;
            poolItem.gameObject = GameObject.Instantiate(poolObject);
            poolItem.gameObject.SetActive(false);
            poolItemList.Add(poolItem);
        }
    }
```

결과 및 분석

MemeoryPool 스크립트 코드2

```
    public GameObject ActivatePoolItem(GameObject poolObject)
    {
        if (poolItemList ==null)
        {
            return null;
        }
        //현재 생성해서 관리는 모든 게임 오브젝트 개수와 현재 활성화된 상태의 게임 오브젝트 비교
        if (MaxCount == activeCount)
        {
            InstantiateObject(poolObject);
        }
        for (int i =0;i < poolItemList.Count;i++)
        {
            PoolItem poolItem = poolItemList[i];
            if (poolItem.gameObject.activeSelf ==false) //게임 오브젝트가 비활성화되어있는지 확인하는 프로퍼티
            {
                activeCount++;
                poolItem.gameObject.SetActive(true);
                return poolItem.gameObject;
            }
        }
        return null;
    }
    public void DeactivatePoolItem(GameObject removeObject)
    {
        if(poolItemList ==null || removeObject ==null)
        {
            return;
        }
        for(int i =0; i< poolItemList.Count ;i++)
        {
            PoolItem poolItem = poolItemList[i];
            if (poolItem.gameObject == removeObject)
            {
                activeCount--;
                poolItem.gameObject.SetActive(false);
                return;
            }
        }
    }
}
```

코드 해석

MemeoryPool(GameObject poolObject) 생성자: 이 생성자는 메모리 풀을 초기화하고, 첫 번째 게임 오브젝트를 생성합니다.

InstantiateObject(GameObject poolObject): 이 메소드는 새로운 게임 오브젝트를 생성하고, 이를 메모리 풀에 추가합니다. 생성된 게임 오브젝트는 초기에 비활성화 상태로 설정됩니다.

ActivatePoolItem(GameObject poolObject): 이 메소드는 비활성화된 게임 오브젝트 중 하나를 활성화하고 반환합니다. 만약 모든 게임 오브젝트가 활성화되어 있다면, 새로운 게임 오브젝트를 생성합니다.

DeactivatePoolItem(GameObject removeObject): 이 메소드는 주어진 게임 오브젝트를 메모리 풀에서 비활성화합니다.

메모리 풀을 사용함으로써 게임에서 자주 생성하고 파괴되는 게임 오브젝트를 효율적으로 관리하는 데 사용할 수 있습니다.

결과 및 분석

CreatUnit 스크립트 코드

```
public class CreatUnit : MonoBehaviour
{
    private MemeoryPool memeoryPool;
    private string[] unitName =new string[2];
    private void Awake() //신이 초기화 될때 미리 메모리 풀을 생성하기위해 사용
    {
        unitName[0] ="Goblin";
        unitName[1] ="Slime";
        memeoryPool =new MemeoryPool(Resources.Load<GameObject>(unitName[Random.Range(0, 2)]));
    }
    // Start is called before the first frame update
    void Start()
    {
        StartCoroutine(Create());
    }
    IEnumerator Create()
    {
        var wait =new WaitForSeconds(3f);
        while (true)
        {
            yield return wait;
            GameObject monster = Resources.Load<GameObject>(unitName[Random.Range(0, 2)]);
            monster = memeoryPool.ActivatePoolItem(monster);
            monster.GetComponent<Monster>().SetUp(memeoryPool);
            monster.transform.position = Random.insideUnitCircle.normalized *7.5f;
        }
    }
}
```

코드 해석

'Awake()' 함수: 이 함수는 유니티 엔진에서 제공하는 함수로, 게임 객체가 생성될 때 한 번만 호출됩니다. 여기서는 메모리 풀을 초기화하고, 생성할 몬스터 유닛의 이름을 설정합니다.

'Start()' 함수: 이 함수는 게임 오브젝트가 활성화될 때 한 번만 호출되며, 'Create()' 코루틴을 시작합니다.

'Create()' 코루틴: 이 코루틴은 무한히 반복되며, 매번 3초마다 새로운 몬스터 유닛을 생성합니다. 생성된 몬스터 유닛은 메모리 풀에서 활성화되고, 메모리 풀을 참조할 수 있도록 설정됩니다. 그리고 몬스터 유닛의 위치는 원의 경계 내에서 무작위로 결정됩니다.

CreatUnit 스크립트는 게임에 몬스터 유닛을 무작위로 생성하는 역할을 합니다. 메모리 풀을 사용함으로써, 게임 오브젝트를 효율적으로 재사용할 수 있습니다.

결과 및 분석

핵심적인 내용은 몬스터 출몰(생성) 위치
monster.transform.position = Random.insideUnitCircle.normalized * 7.5f;
몬스터의 위치는 반지름이 7.5인 원의 경계 위의 무작위한 위치로 생성되게 하는 코드이며 무작위 위치의 원 안에서 몬스터가 생성하여 나오게 만드는 식입니다.

3. 에셋 스토어에 다운받은 몬스터 스프라이트를 prefab으로 만들어 줍니다.

4. Moster 스크립트와 Slime, Goblin 스크립트 생성합니다.

5. Slime, Goblin스크립트는 Moster 상속 받게 설정해줍니다.
각각 Slime, Goblin의 기본정보를 입력해줍니다.

5-1. Slime 스크립트 기본정보

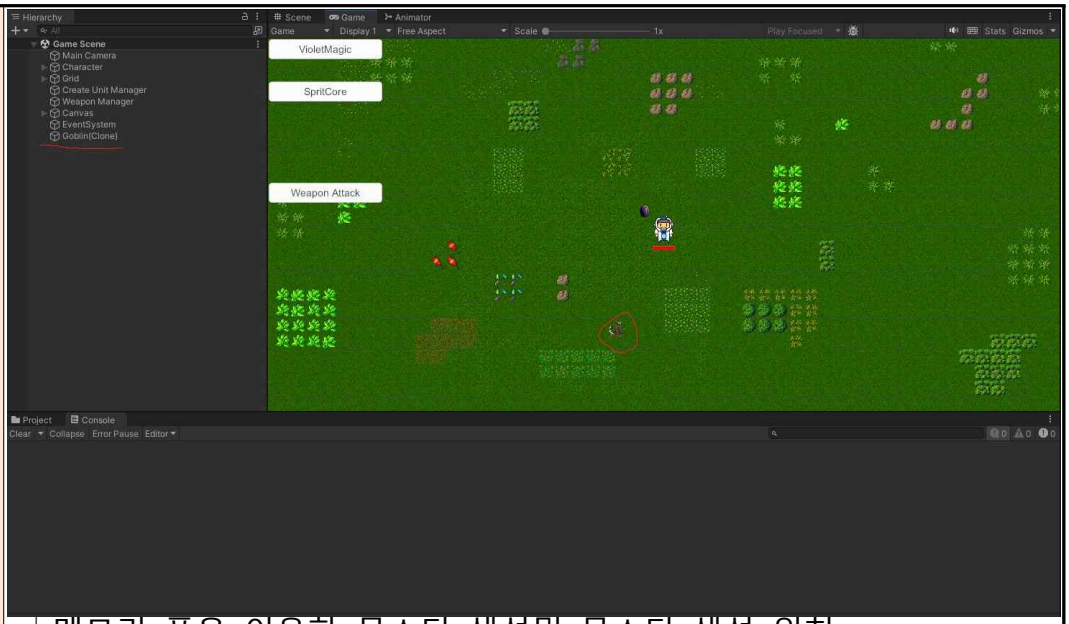
```
protected override void Start()
{
    health =50;
    attack =1;
    base.Start();
}
```

5-2. Goblin 스크립트 기본정보

```
protected override void Start()
{
    health =100;
    attack =5;
    base.Start();
}
```

여기서 base.Start();는 코드는 상속받은 부모 클래스인 Monster의 Start 메소드를 호출하는 역할을 합니다.
즉, Slime 클래스가 Monster 클래스를 상속받았으므로, base.Start();를 통해 Monster 클래스에 정의된 Start 메소드의 기능을 실행하게 됩니다.

결과 및 분석



↳메모리 풀을 이용한 몬스터 생성및 몬스터 생성 위치

- **무작위 몬스터와 메인 캐릭터(플레이어)가 만날 시 충돌처리 과정 구현**

1. 캐릭터에 box collider2D를 생성합니다.

box collider2D- 2D 물리 시스템과 상호 작용하는 콜라이더입니다.

2. prefab에 있는 몬스터한테도 box collider2D를 생성합니다.

3. player(메인 캐릭터 설정) 스크립트에 코드 추가

```
public void OnTriggerEnter2D(Collider2D collision)
{
    IAttack obj = collision.GetComponent<IAttack>();
    var monster = collision.GetComponent<Monster>();
    if (obj !=null)
    {
        Damage(monster);
    }
    Debug.Log("공격받은 나의 체력:"+ health);
}
```

코드 해석

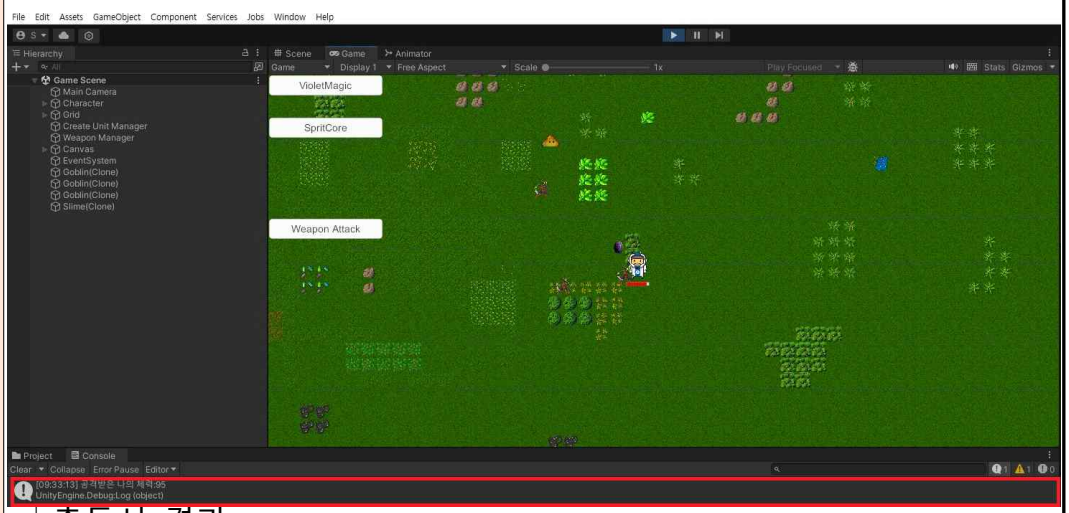
이 함수는 몬스터가 다른 오브젝트와 충돌했을 때 호출되는 함수입니다. 'OnTriggerEnter2D()'는 유니티에서 제공하는 함수로, 충돌한 오브젝트의 Collider2D 컴포넌트를 인자로 받습니다.

충돌한 오브젝트에서 'IAttack' 인터페이스를 구현한 컴포넌트와 'Monster' 클래스를 구현한 컴포넌트를 가져옵니다.

만약 충돌한 오브젝트가 'IAttack' 인터페이스를 구현하고 있다면, 그 오브젝트로부터 'Damage()' 함수를 호출하여 몬스터에게 피해를 입힙니다. 이 때, 피해를 입히는 오브젝트는 'monster' 변수에 저장된 몬스터입니다.

마지막으로 'Debug.Log()' 함수를 통해 몬스터의 현재 체력을 로그로 출력합니다.
이 함수는 몬스터가 다른 오브젝트와 충돌했을 때 그 오브젝트가 공격 가능한 오브젝트인지 확인하고, 공격 가능한 오브젝트라면 피해를 입히는 로직을 구현하고 있습니다. 'IAttack' 인터페이스를 구현한 오브젝트는 공격 가능한 오브젝트로 간주된다는 것을 알 수 있습니다.

결과 및 분석



↳충돌시 결과

- **무작위 몬스터가 플레이어를 따라가는 기능 구현**

1.부모 클래스인 Moster 스크립트 클래스 코드 추가

```
protected virtual void Start()
{
    speed =1.0f;
    spriteRenderer = GetComponent<SpriteRenderer>();
    player = GameObject.Find("Character").GetComponent<Transform>();
}
```

코드 해석

몬스터의 초기 속도를 설정하고, 스프라이트 렌더러와 플레이어의 위치를 가져오는 작업을 수행합니다.

'speed = 1.0f;': 이 몬스터의 속도를 1.0f로 초기화합니다.

'spriteRenderer = GetComponent<SpriteRenderer>();': 이 몬스터가 속한 게임 오브젝트의 SpriteRenderer 컴포넌트를 가져와 spriteRenderer 변수에 할당합니다. 이를 통해 이후에 스프라이트의 속성을 변경할 수 있습니다.

'player = GameObject.Find("Character").GetComponent<Transform>();': "Character"라는 이름의 게임 오브젝트를 찾아 그 오브젝트의 Transform 컴포넌트를 가져와 player 변수에 할당합니다. 이를 통해 플레이어 캐릭터의 위치 정보에 접근할 수 있습니다.

'virtual' 키워드는 이 메소드가 하위 클래스에서 재정의(override)될 수 있음을 나타냅니다. 이는 각기 다른 세부적인 행동을 가진 여러 몬스터를 구현할 때 유용합니다.

결과 및 분석

2. 상속 받고 있는 각각의 Slime, Goblin 스크립트에 밑에 코드 추가

transform.position = Vector3.MoveTowards(transform.position, player.position, Time.deltaTime * speed);

↳위의 코드는 현재 오브젝트를 프레임마다 플레이어의 위치로 속도에 따라 이동시키는 역할을 합니다.

3. 방향전환(캐릭터 움직임 코드와 똑같은 개념으로 방향 전환) 코드 추가

```
direction = transform.position - player.position;

if (direction.x > 0)
{
    spriteRenderer.flipX = true;

}
else if (direction.x < 0)
{
    spriteRenderer.flipX = false;
}
```

코드 해석

핵심은 direction = transform.position - player.position;=몬스터의 현재 위치에서 플레이어의 위치를 빼서, 몬스터가 플레이어에게서 어느 방향에 있는지를 계산해줍니다.

1.만약 몬스터가 플레이어의 오른쪽에 있다면 (즉, direction.x가 0보다 크다면), 스프라이트를 x축으로 뒤집습니다. 이는 몬스터가 플레이어를 바라보도록 합니다.

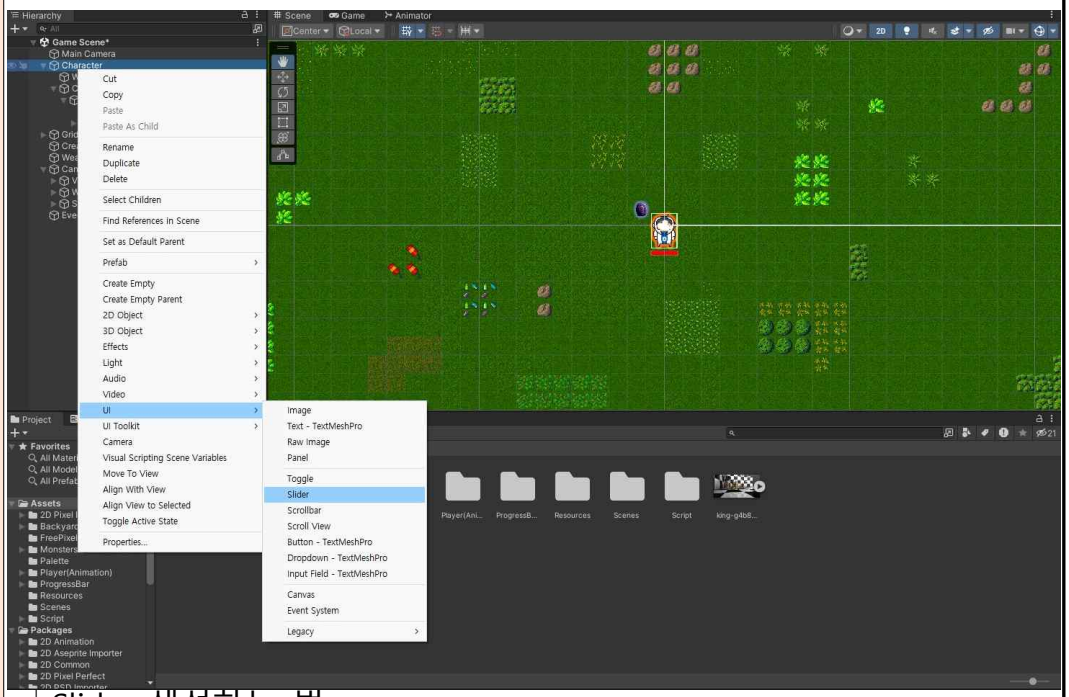
2.만약 몬스터가 플레이어의 왼쪽에 있다면 (즉, direction.x가 0보다 작다면), 스프라이트를 뒤집지 않습니다. 이는 몬스터가 이미 플레이어를 바라보도록 합니다.

위의 코드로 몬스터가 캐릭터를 따라오고 캐릭터 위치마다 방향전환을 할 수 있게 만들어 줍니다.

결과 및 분석

● **메인 캐릭터(플레이어) 체력 바 생성**

1. Scene에 있는 Character에 자식으로 UI->Slider를 추가 합니다.



↳Slider 생성하는 법

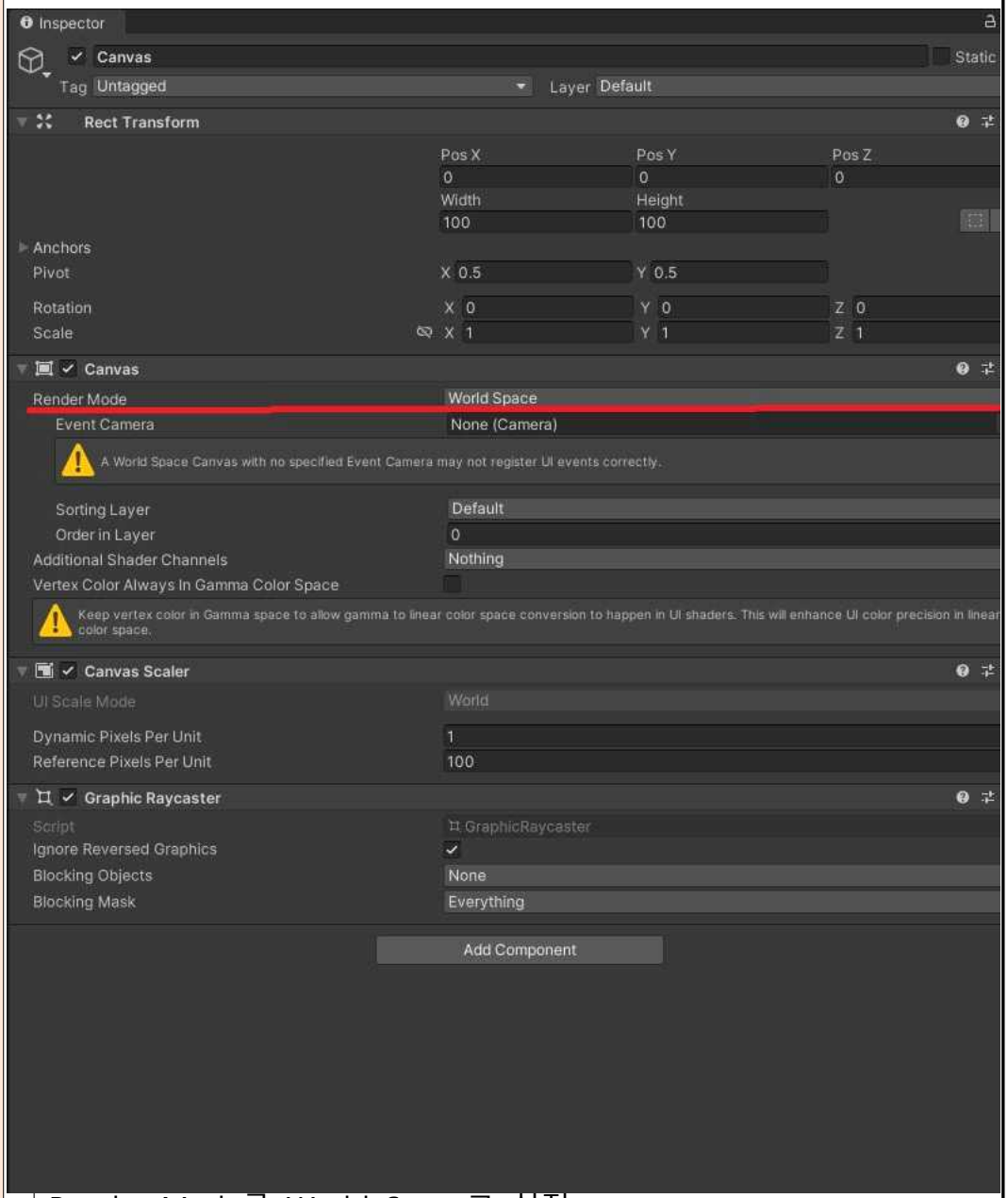
2. 바로 Character(플레이어)에 상속하면 Slider가 따라가는데 딜레이가 걸립니다.

해결 방법-Render Mode World Space로 설정합니다.

원리-'World Space' 모드에서 UI 요소는 일반적인 게임 오브젝트와 같이 취급되며, 그 위치, 회전, 크기는 세계 좌표계에 따라서 결정됩니다.

이는 UI 요소를 3D 게임 세계 안에 자연스럽게 통합하거나, 특정 게임 오브젝트에 UI 요소를 부착해야 하는 경우에 유용합니다.

결과 및 분석



↳Render Mode를 World Space로 설정

3. 크기를 설정해주고 슬라이더 바의 위치를 조정해줍니다.

4. Slider에 상속 받아 있는 Fill의 Color를 원하는 색으로 변경합니다.
****체력바를 표시하는 방법은 Slider의 Value 값을 조정해서 표시 합니다.**
Value 값이 1이되면 색이 다 나오고 0이 되면 색이 점차 없어지는 형식입니다.

결과 및 분석

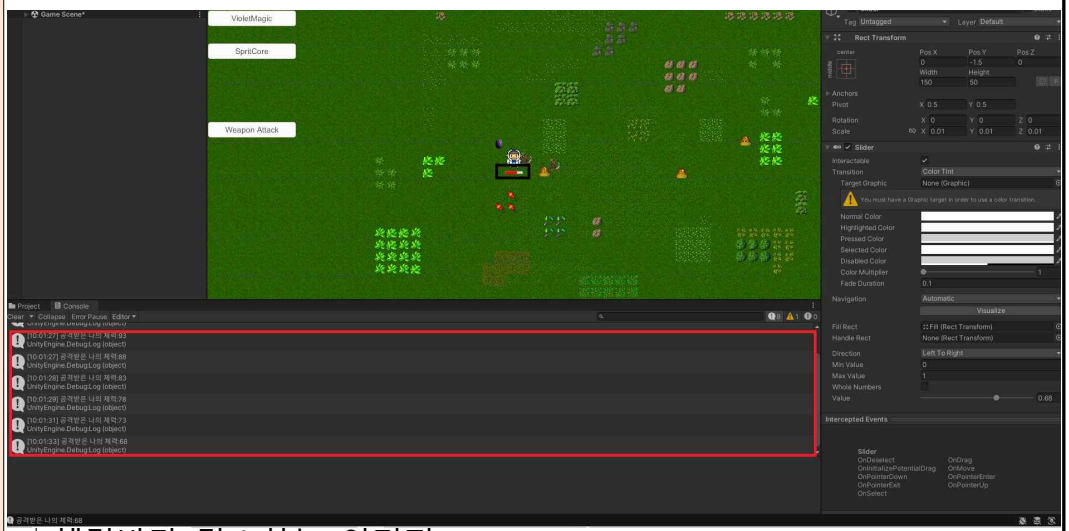
5. Player 스크립트 코드 추가

```
public class Player : MonoBehaviour
{
[SerializeField] float health =100;
[SerializeField] Slider healthSlider;
void Start()
    {
        maxHealth = health;
    }
    public void Damage(Monster monster)
    {
        health -= monster.attack;
        healthSlider.value = health/maxHealth;
    }
}
```

코드 해석

Damage(Monster monster) 함수: 이 함수에서는 플레이어가 몬스터로부터 데미지를 받는 것을 처리합니다. 플레이어의 체력은 몬스터의 공격력만큼 감소하며, 체력 슬라이더의 값도 갱신됩니다.

healthSlider.value = health/maxHealth;의 코드는 Slider의 Value 값이 1이기 때문에 1에 맞춰서 값이 나와야 하므로 값을 나누는 겁니다.



↳체력바가 감소하는 이미지

위 이미지와 같이 체력 값이 감소 할 때마다 슬라이더 value값이 나아져 체력바가 사라지는 모습을 확인 할 수 있습니다.

결과 및 분석

● **버튼 기능을 통한 무기를 바꾸는 기능 추가**

1.에셋 스토어에 원하는 무기나 아이템 UI를 다운 받습니다.

2. IAttack스크립트에 코드 추가합니다.

```
//인터페이스를 쓰기위한 규칙 I가 붙어야함(무조건)
public interface IAttack
{
    void Use();
}
public interface IWeapon
{
    public void Attack();
}
```

코드해석

인터페이스는 클래스나 구조체가 특정한 메소드를 반드시 구현하도록 강제하는 규칙을 정의하는 역할을 합니다.

IAttack 인터페이스: 이 인터페이스는 Use 메소드를 정의하고 있습니다. 이를 구현하는 클래스는 Use 메소드를 반드시 구현해야 합니다.

IWeapon 인터페이스: 이 인터페이스는 Attack 메소드를 정의하고 있습니다. 이를 구현하는 클래스는 Attack 메소드를 반드시 구현해야 합니다.

이런 방식으로 인터페이스를 사용하면, 다른 클래스에서 이러한 인터페이스를 구현하면서 동일한 동작을 수행할 수 있습니다. 예를 들어, IWeapon 인터페이스를 구현하는 클래스는 Attack 메소드를 구현하여 자신의 공격 동작을 정의할 수 있습니다. 이를 통해 코드의 일관성과 재사용성을 높일 수 있습니다.

3. Weapon이라는 스크립트 생성

```
public class Weapon
{
    private IWeapon weapon;
    // Start is called before the first frame update
    public void SetWeapon(IWeapon weapon)
    {
        this.weapon = weapon;
    }
    public void Attack()
    {
        weapon.Attack();
    }
}
```

결과 및 분석

코드해석

private IWeapon weapon: 이 변수는 IWeapon 인터페이스를 참조합니다. IWeapon 인터페이스를 구현하는 어떤 객체라도 이 변수에 할당될 수 있습니다.

public void SetWeapon(IWeapon weapon): 이 메소드는 IWeapon 인터페이스를 구현하는 객체를 weapon 변수에 설정하는 역할을 합니다. 이 메소드를 통해 원하는 무기를 동적으로 변경할 수 있습니다.

public void Attack(): 이 메소드는 weapon 변수가 참조하는 객체의 Attack 메소드를 호출합니다.

이 클래스를 사용하면, SetWeapon 메소드를 통해 어떤 무기를 사용할지 동적으로 결정하고, Attack 메소드를 통해 해당 무기의 공격 동작을 수행할 수 있습니다. 이를 통해 다양한 무기를 쉽게 관리하고 변경할 수 있습니다. 이는 게임에서 무기 변경 기능 등을 구현할 때 유용합니다.

4.Weapon Manager 스크립트 생성

```
public class WeaponManager : MonoBehaviour
{
private Weapon weapon;
[SerializeField] SpriteRenderer spriteRenderer;
void Start()
{
weapon =new Weapon();
    spriteRenderer.sprite = Resources.Load<Sprite>("VioetMagic");
    weapon.SetWeapon(new VioetMagic());
}
public void ChangeMagic()
{
    spriteRenderer.sprite = Resources.Load<Sprite>("VioetMagic");
    weapon.SetWeapon(new VioetMagic());
}
public void Changecore()
{
    spriteRenderer.sprite = Resources.Load<Sprite>("SpritCore");
    weapon.SetWeapon(new SpritCore());
}
public void Attack()
{
    weapon.Attack();
}
void Update()
{

}
}
```

결과 및 분석

코드해석

private Weapon weapon: 이 변수는 Weapon 클래스의 인스턴스를 참조합니다. 이 변수를 통해 무기의 공격 기능을 사용할 수 있습니다. Start() 함수: 이 함수에서는 Weapon 객체를 생성하고, 스프라이트 렌더러에 'VioetMagic' 스프라이트를 로드합니다. 그리고 weapon 객체에 'VioetMagic' 무기를 설정합니다.

ChangeMagic() 함수: 이 함수에서는 스프라이트 렌더러에 'VioetMagic' 스프라이트를 로드하고, weapon 객체에 'VioetMagic' 무기를 설정합니다. 이 함수를 호출하면 현재 무기를 'VioetMagic'으로 변경할 수 있습니다.

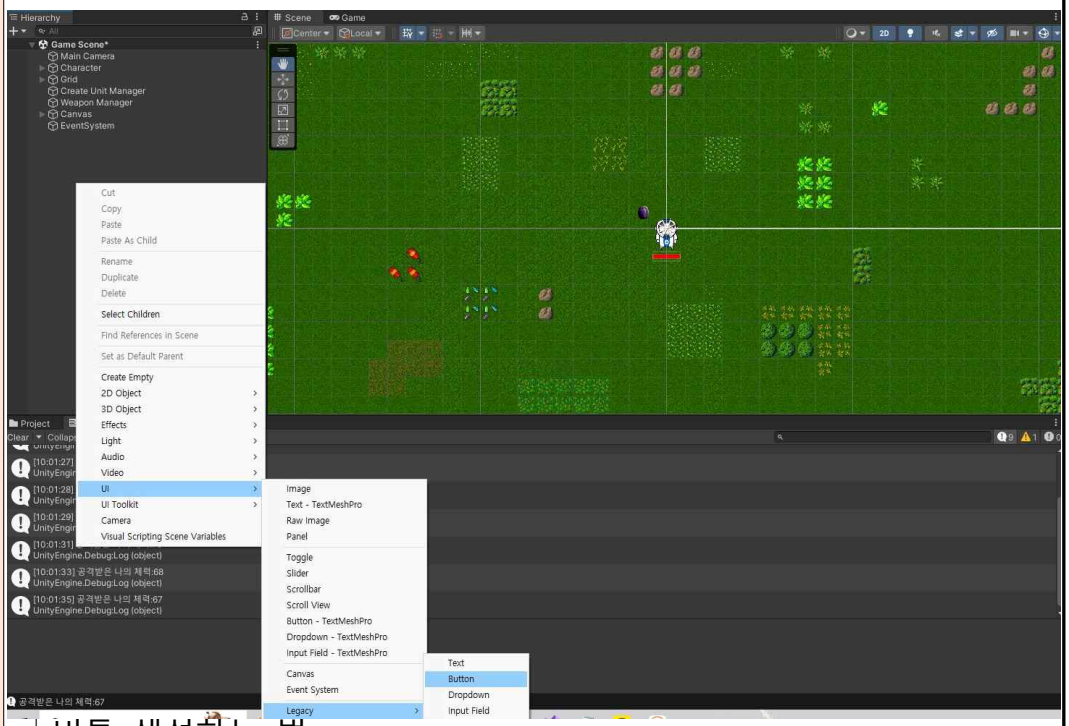
Changecore() 함수: 이 함수에서는 스프라이트 렌더러에 'SpritCore' 스프라이트를 로드하고, weapon 객체에 'SpritCore' 무기를 설정합니다. 이 함수를 호출하면 현재 무기를 'SpritCore'으로 변경할 수 있습니다.

Attack() 함수: 이 함수에서는 weapon 객체의 'Attack' 메소드를 호출합니다. 이 함수를 호출하면 현재 설정된 무기로 공격할 수 있습니다.

이 클래스를 사용하면, 무기를 변경하고, 공격하는 기능을 쉽게 구현 가능 합니다.

5. Weapon Manager라는 빈 오브젝트(Create Empty)를 Scene에 추가 및 오브젝트에 Weapon Manager스크립트 추가합니다.

6. 버튼을 생성하고 위치를 조정 해줍니다.

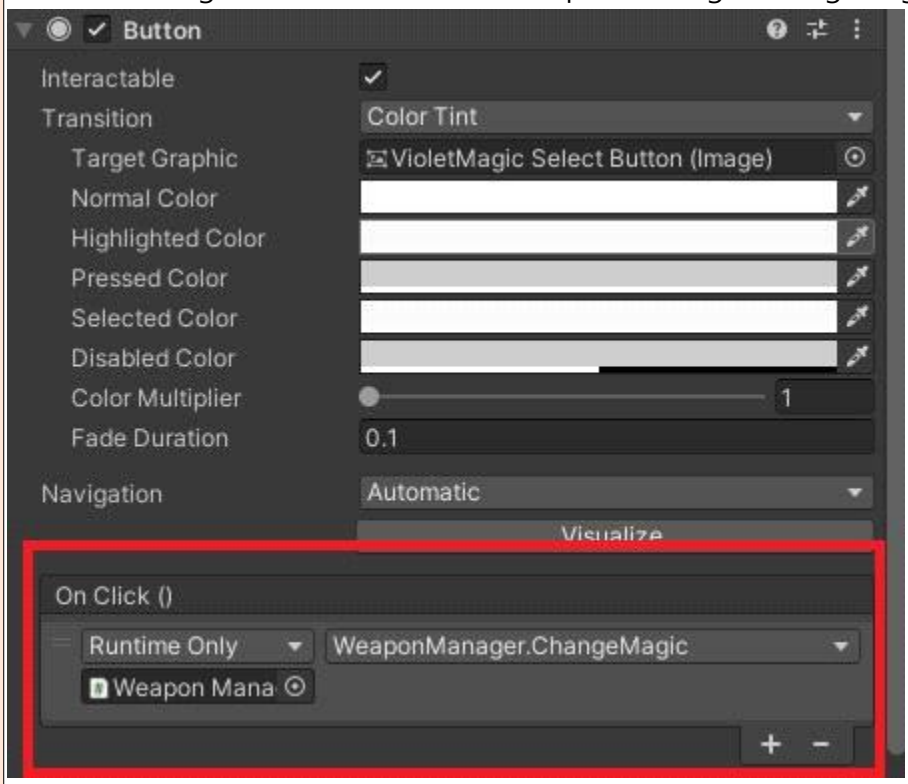


↳버튼 생성하는 법

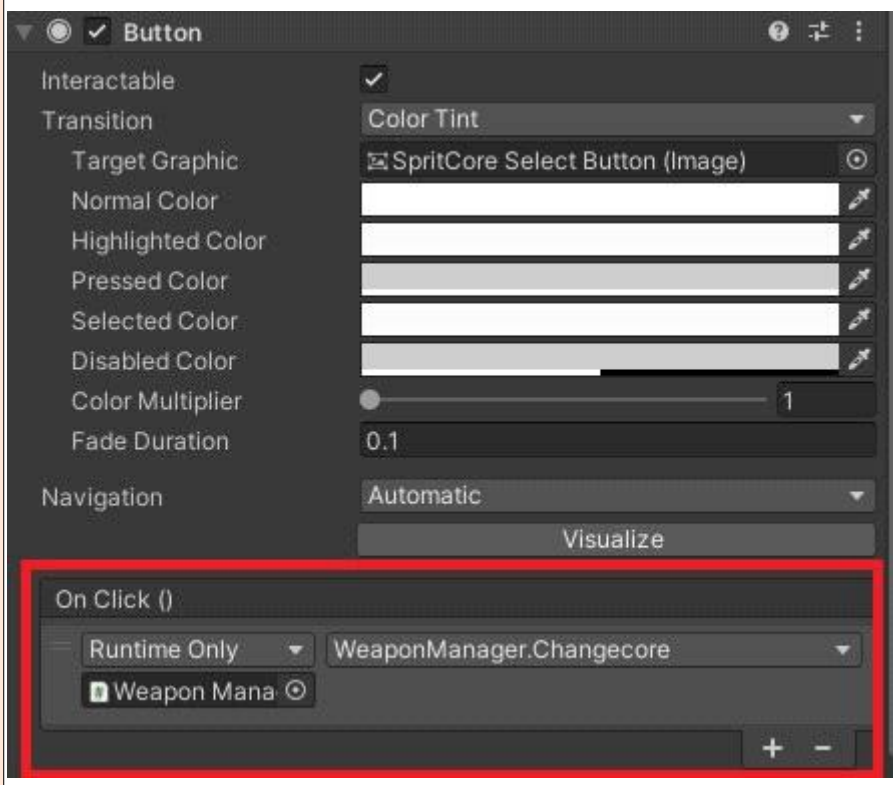
결과 및 분석

7. 각 버튼의 onClick을 추가해 WeaponManager를 추가합니다.
Weapon Manager에서 설정한 코드에 따라 지정해줍니다.

7-1 VioletMagic Select Button에는 Weapon Manager.ChangeMagic을 설정

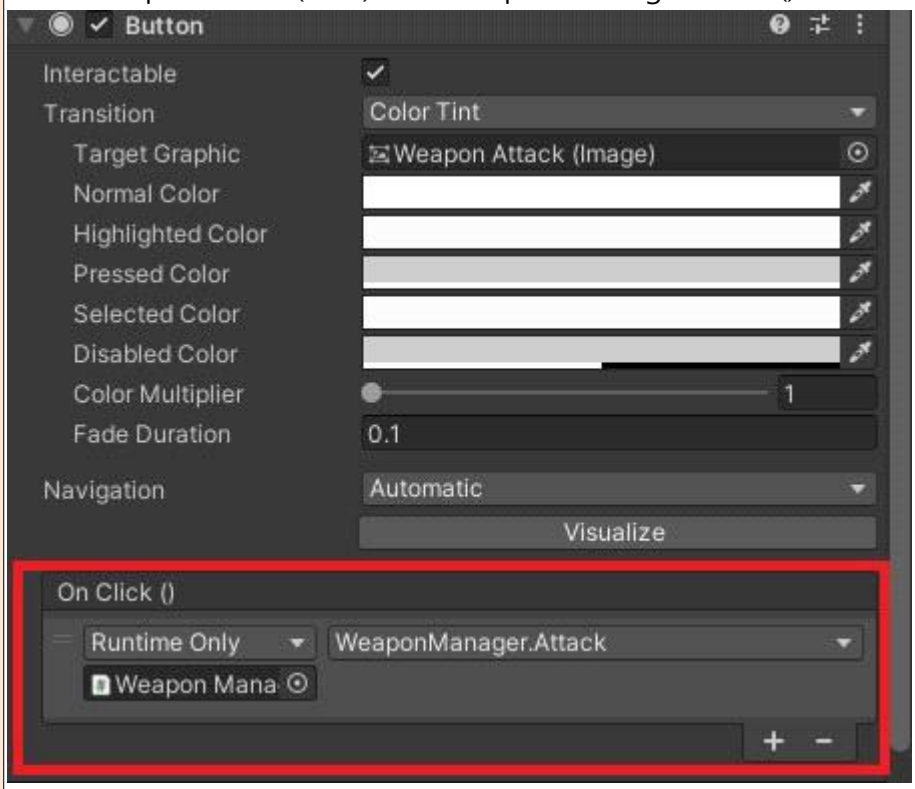


7-2 SpritCore Select Button에는 Weapon Manager.Changecore()을 설정



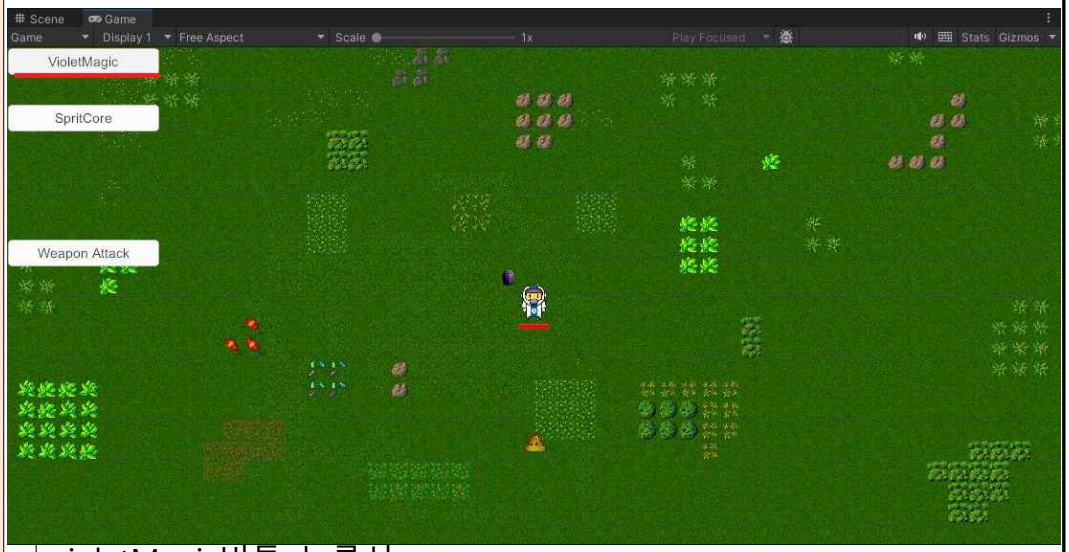
결과 및 분석

7-3. Weapon Attack(버튼)에는 Weapon Manager.Attack()을 설정

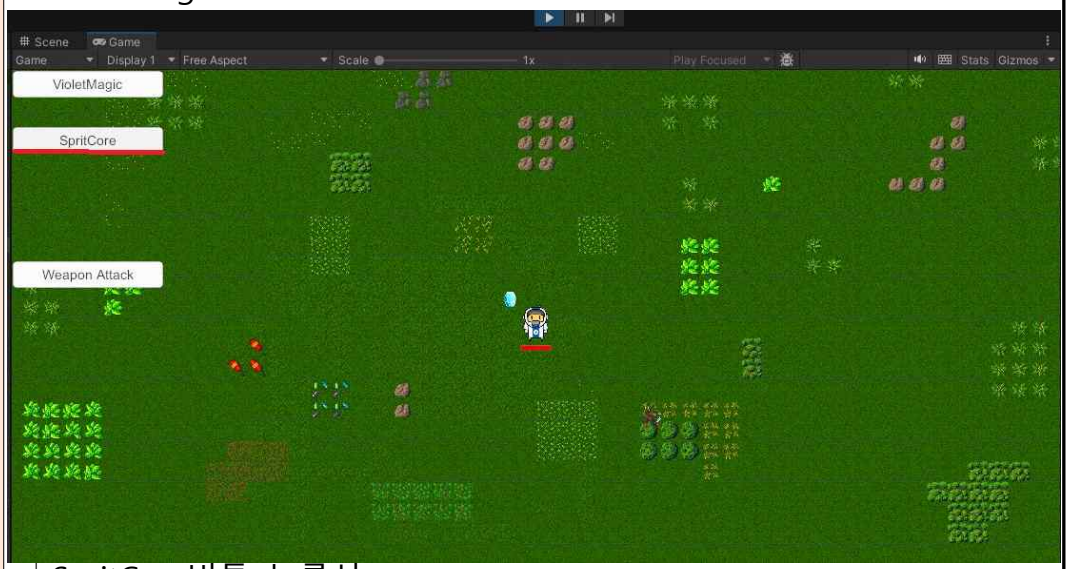


결과 및 분석

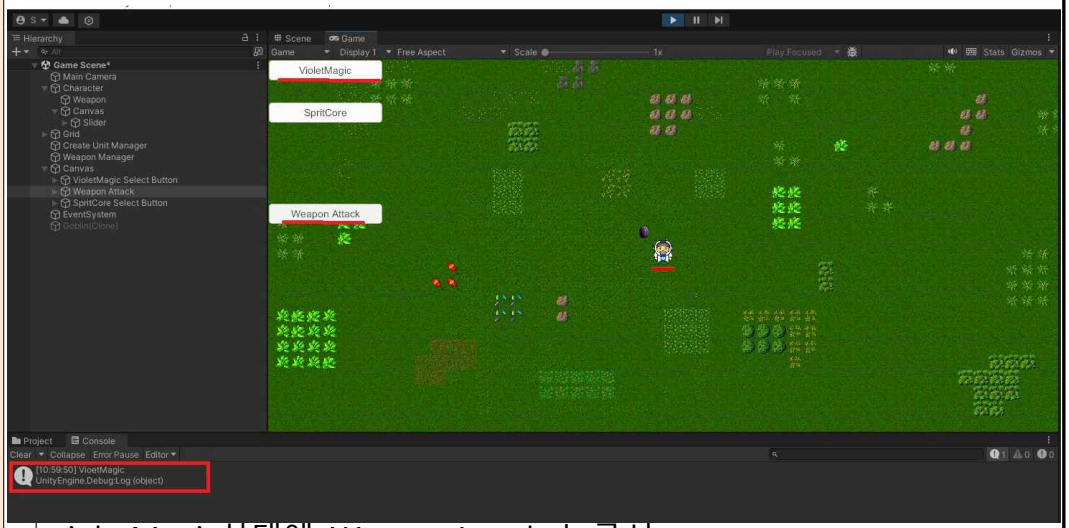
8. 결과 사진



↳violetMagic버튼 누를시



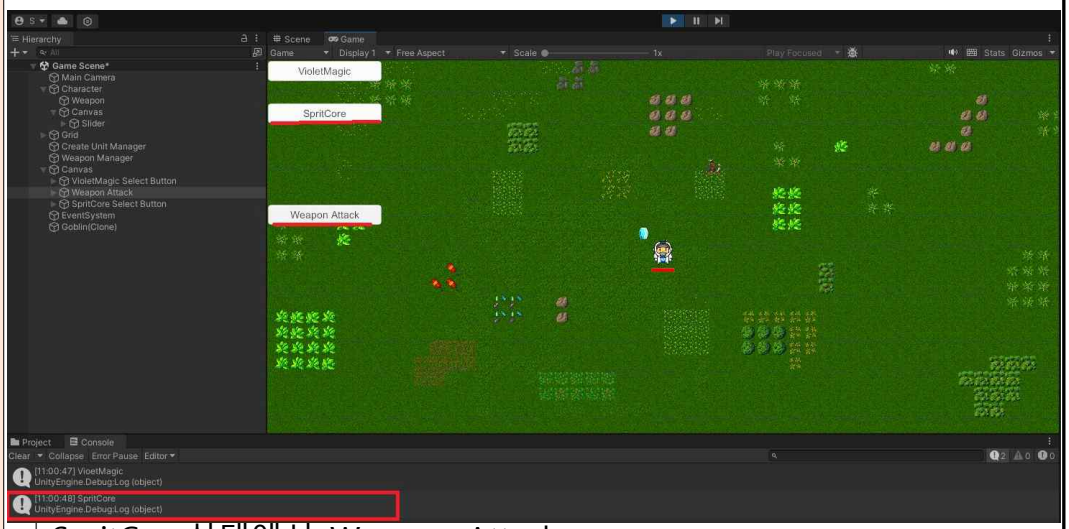
↳SpritCore버튼 누를시



↳violetMagic상태에 WeaponAttack 누를시

결과 및 분석

8. 결과 사진 2



↳SpritCore상태에서 Weapon Attack

〈 참고 문헌 또는 리소스: URL, 도서 등 〉

(!! 스스로 작성하지 않은 리소스 출처는 반드시 밝힐 것 !!)

(참고 내용은 접근 방법란에 명시할 것)

1)

https://assetstore.unity.com/packages/2d/characters/2d-character-astronaut-182650

https://assetstore.unity.com/packages/2d/environments/backyard-top-down-tileset-53854

https://assetstore.unity.com/packages/2d/characters/free-pixel-mob-113577

https://velog.io/@linkedlist97/%EB%A9%94%EB%AA%A8%EB%A6%AC-%ED%92%80%EC%9D%98-%EC%82%AC%EC%9A%A9