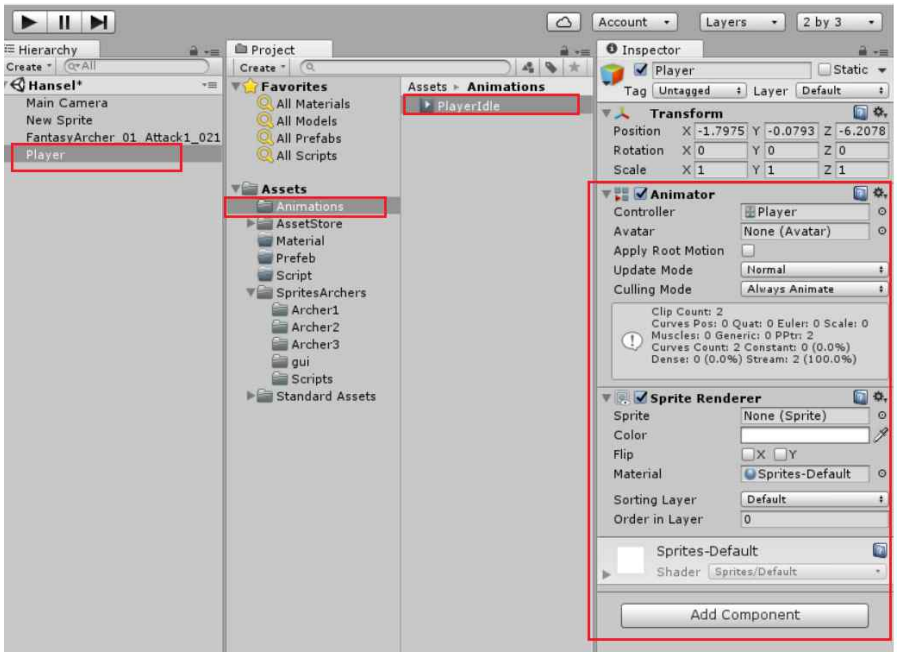
유니티 2D

유니티 실행-씬-오브젝트-콤포넌트
Component(콤포넌트) / ○ Edit메뉴 - Project Setting - Input -
보고있는 씬 장면으로 카메라 바로 이동 : Ctrl + Shift + F

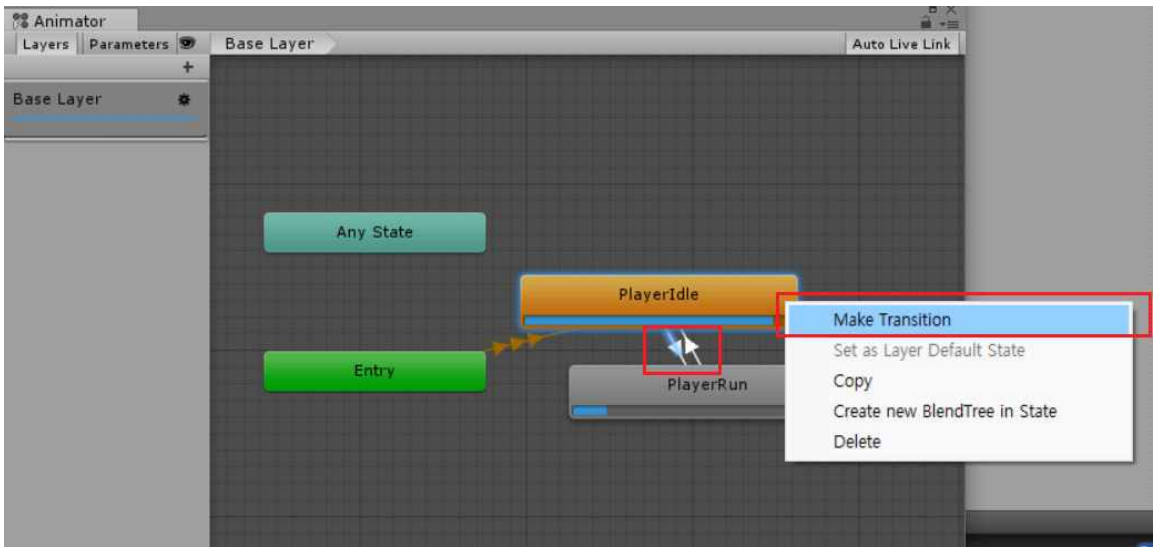
○ 플레이어 생성

1. empty 오브젝트 생성 후 이름을 Player, 한 개씩 잘려져 있는 스프라이트 모음을 끌어서 Prefeb화 함

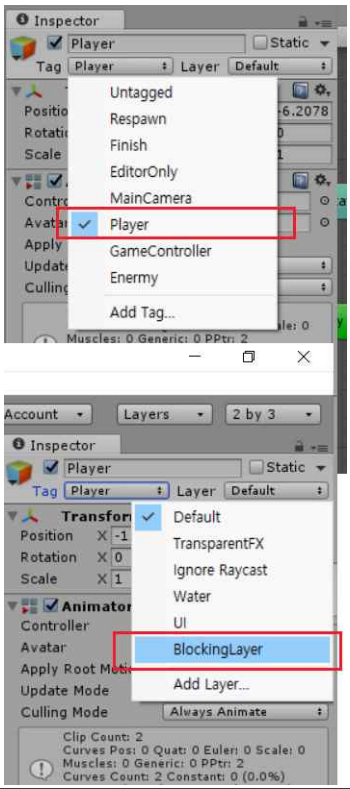
2. 그럼 Player Inspector에 Sprite Renderer 과 Animator이 생성
   - Sprite Renderer : 현재 스프라이트 프레임을 보여줌
   - Animator : 애니메이션이 재생되도록 허가함



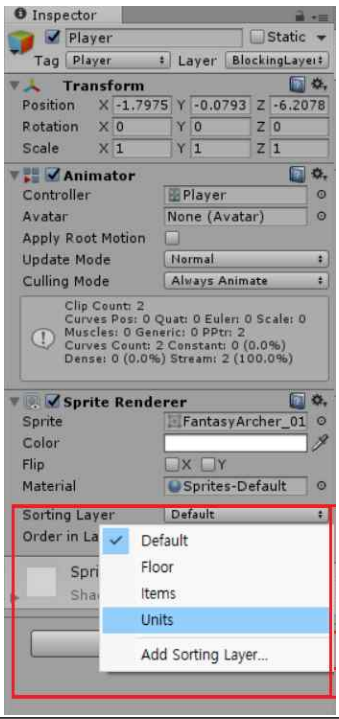
3. Animator에서 Make Transition을 이용하면 상태 변화가 된다.



4. Tag와 Layer를 추가해준다.
   - Player : 플레이어
   - BlockingLayer : 모든 충돌(Collision)들이 체크할 레이어 이다.



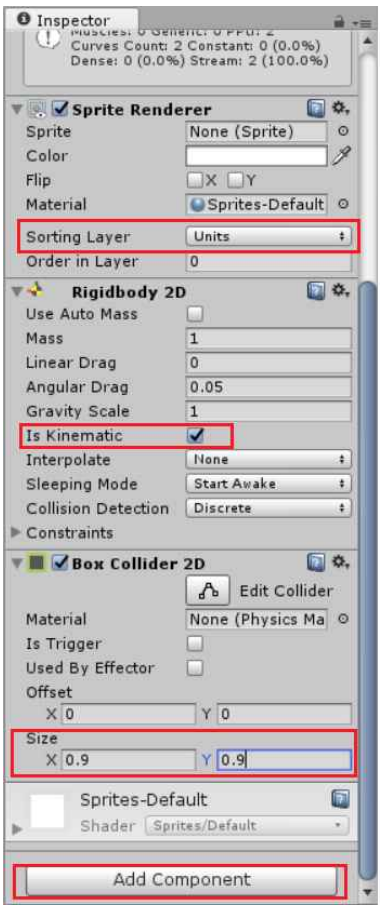
5. 스프라이트 랜더러를 위해 Sorting(정렬) 레이어를 세팅한다.
   - Floor : 배경
   - Items : 기타 아이템 배치
   - Units : 플레이어와 적들의 레이어



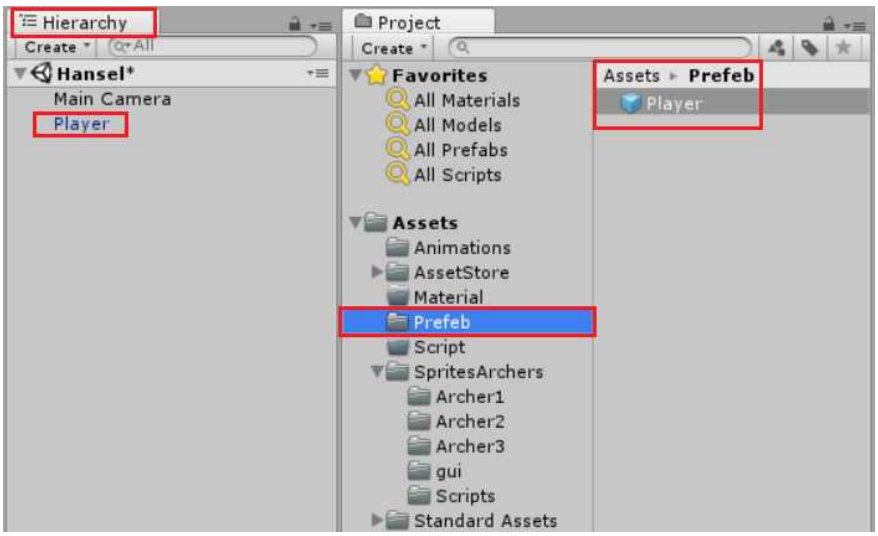
6. Component를 두 가지 추가한다.
   - Box collider 2D (박스 충돌체 2D) : 플레이어와 관련된 충돌을 감지
   - Rigid body 2D : 플레이어를 물리 시스템을 통해 움직이게 함

7. Rigidbody에 Is Kinematic을 설정해준다.
   - Is Kinematic : 플레이어가 이리저리 날라 다니는 것을 막아주고 격자 기반의 공간에서 움직이도록 한다.
     * 스크립트로 컨트롤할 것이기 때문에 체크해준다.

8. BoxCollider에 Size를 설정해준다.
   - 좀 작게 만들어서 인접한 공간의 다른 물건들이 있는 공간으로 움직이려할 때 사고로 부딪치지 않게 해준다.

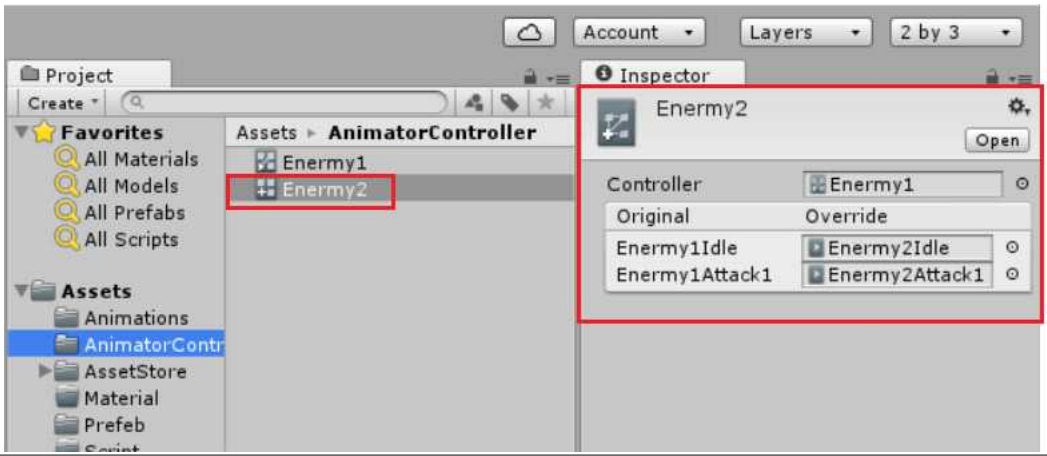


9. Player을 프리팹으로 저장하기 위해 Prefab 폴더로 드래그 한다.
   - 프리팹을 생성하면 Hierarchy(계층)에서 지워도 된다.
   - Hierarchy : 현재 실행중인 오브젝트가 동적으로 존재하는 계층 공간

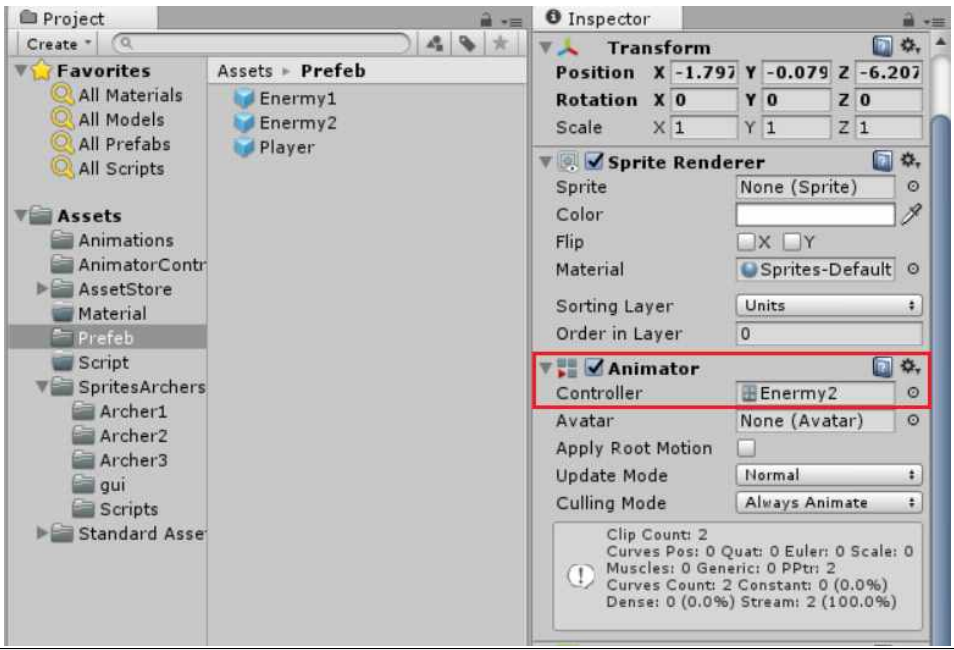


○ 적 생성

1. 플레이어와 동일하게 생성하고, 애니메이션 상태에서 Animator Override Controller를 생성한다.
   - 어떤 컨트롤러를 재정의(Override)할 것인지 정한 뒤 Override 한다.

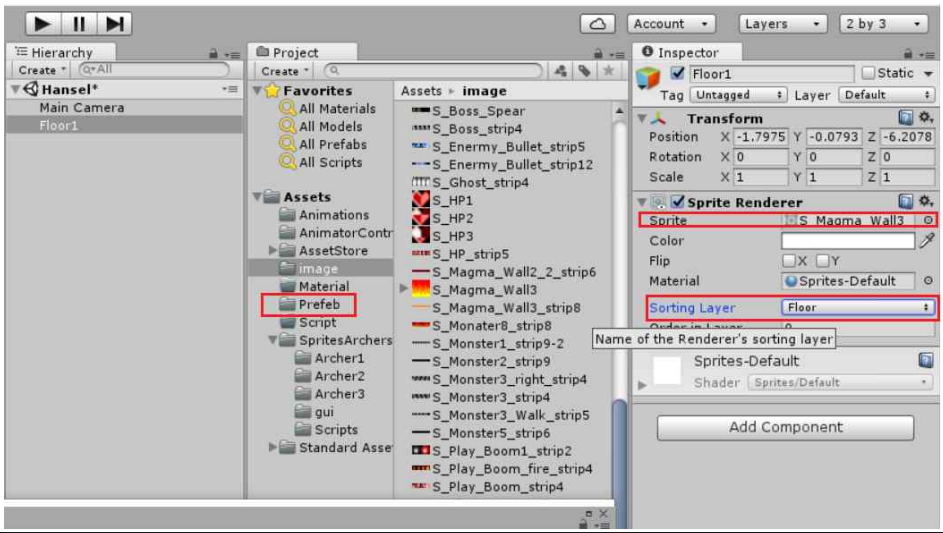


2. Enermy2의 Animator의 Controller를 오버라이드한 Enermy2로 바꾼다.



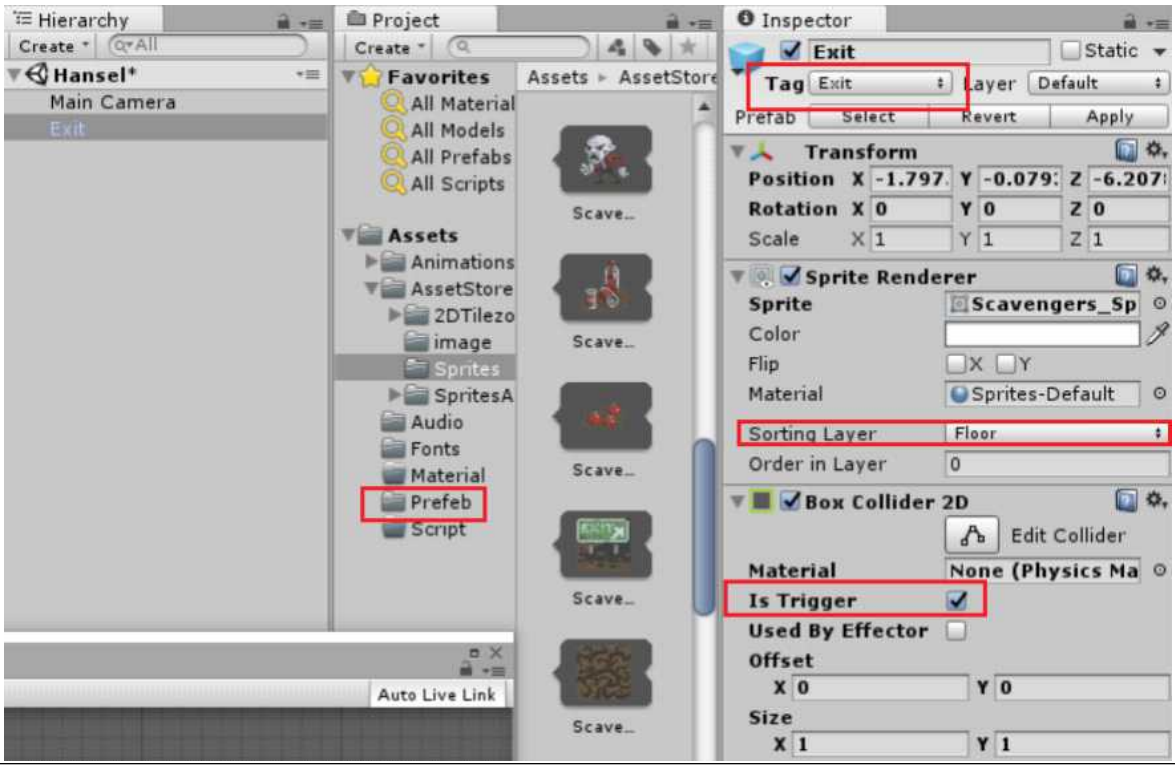
○ 땅 생성

1. EmptyObject를 생성 후 Floor1로 바꾼 뒤 Sprite Renderer를 Component에 추가해준다.
   - Sprite Renderer : 스프라이트를 설정한다.
2. Sprite Renderer의 Sprite를 지정하고, Sorting Layer를 Floor로 바꾼 후 프레펩화 한다.



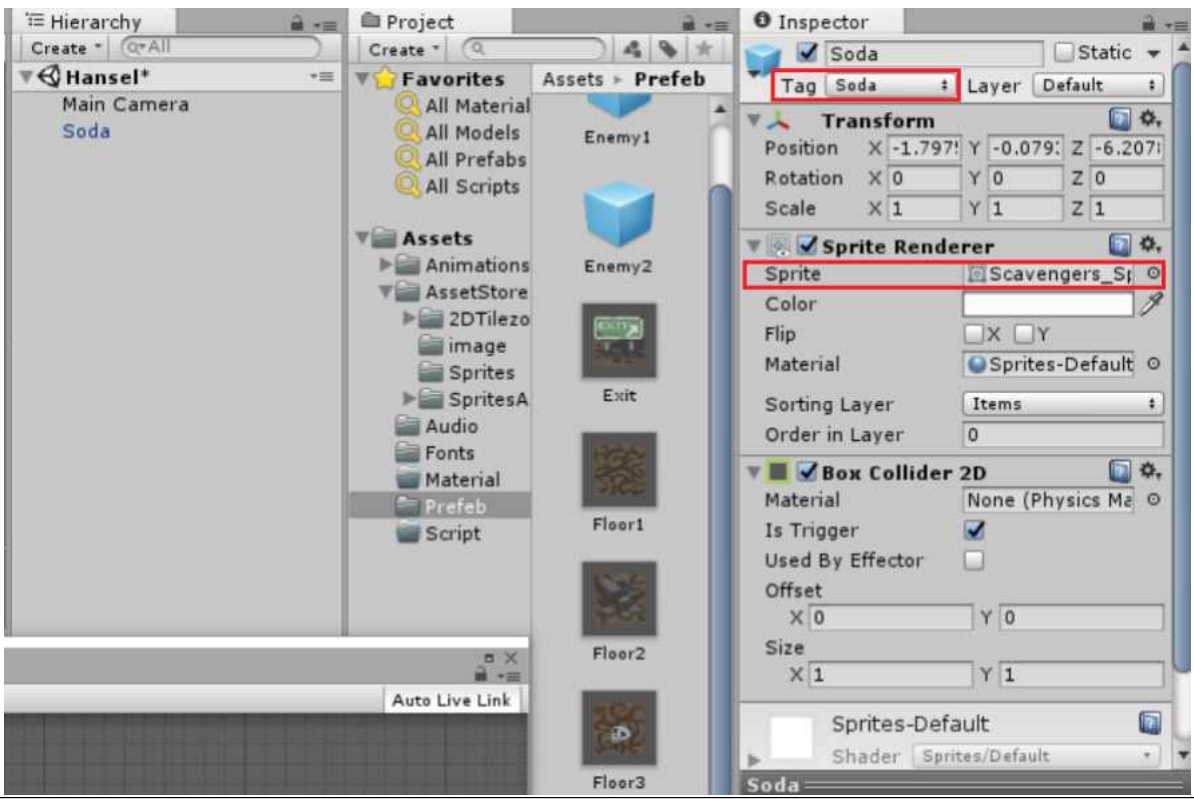
○ 출구 생성

1. 땅의 스프라이트 대신 Exit 스프라이트를 넣고, Box Collider 2D 컴포넌트를 추가한다.
   - Box Collider 2D : 플레이어가 이 곳으로 이동할 때 감지할 수 있다.
2. Box Collider 2D의 Is Trigger을 체크하며, Sorting Layer를 Items로 설정한다.
   - Is Trigger : 플레이어가 이곳으로 이동하는 것을 막지는 않지만, 충돌을 감지하게 할 수 있다.
   - Sorting Layer - Items : 플레이어보다 아래에 그려지면서, 땅보다는 위에 그려진다.
3. 태그를 Exit 태그로 변경한 뒤 프리펩화 시킨다.



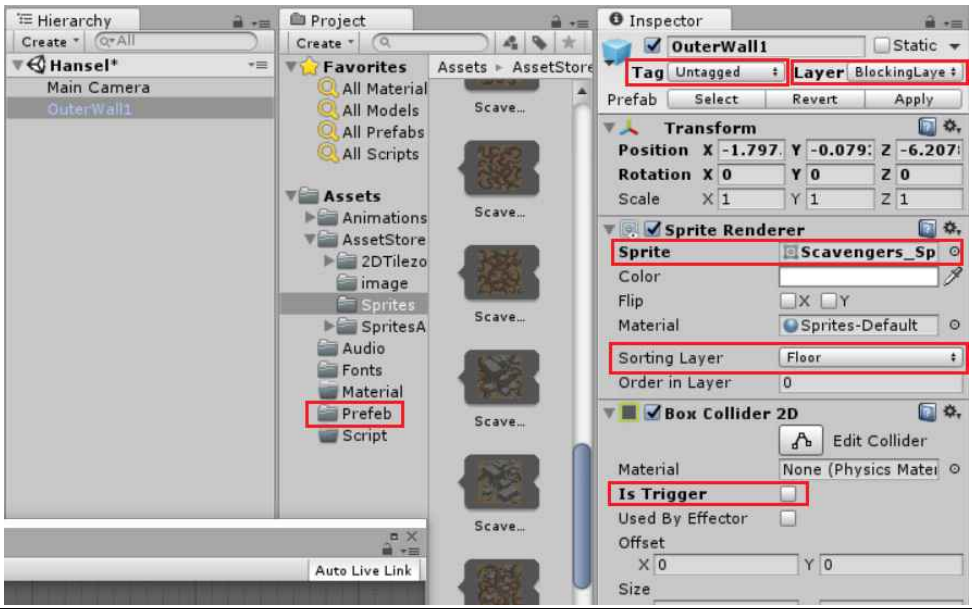
○ 음식 아이템 생성

1. Exit 스프라이트대신 과일 스프라이트를 넣고, 태그를 Food로 변경시키고 프리펩화 한다.



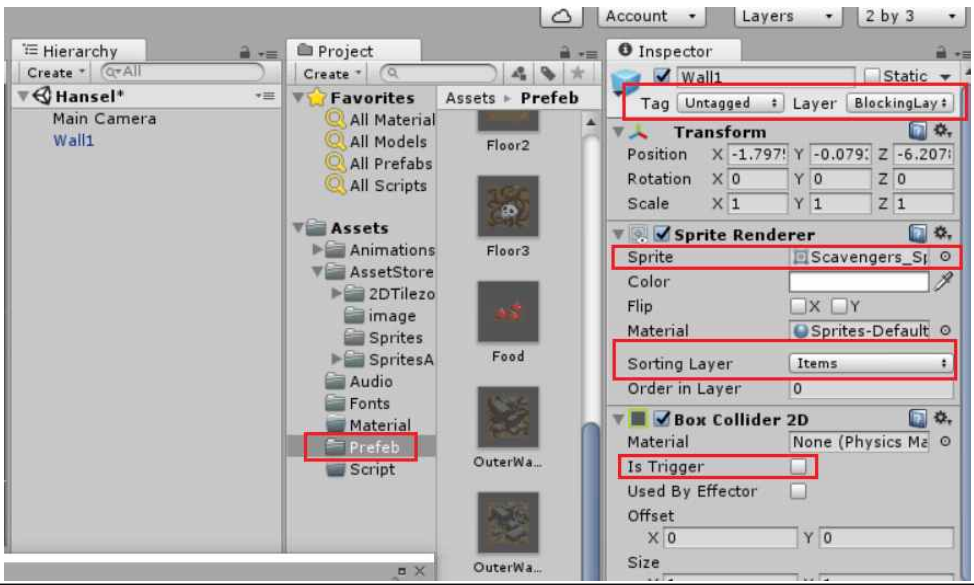
○ 바깥벽(OutWall) 생성

1. 벽 스프라이트를 넣고 Box Collider 2D의 Is Trigger의 체크를 해제한다.
   - Is Trigger : 플레이어나 적이 이곳으로 이동하는 것을 막게 한다.
2. 태그를 Untagged로 설정하고, Layer를 BlockingLayer로 설정한다.
   - BlockingLayer : 실제로 플레이어의 움직임을 막아준다.
3. Sprite Renderer의 Sorting Layer를 Floor로 변경한 뒤 프리펩화 한다.



○ 안쪽 방해벽(Wall) 생성

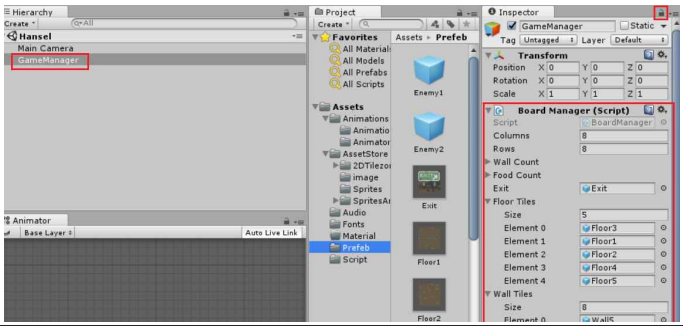
1. 벽 스프라이트를 넣고 Box Collider 2D의 Is Trigger의 체크를 해제한다.
   - Is Trigger : 플레이어나 적이 이곳으로 이동하는 것을 막게 한다.
1. 태그를 Untagged로 설정하고, Layer를 BlockingLayer로 설정한다.
   - BlockingLayer : 실제로 플레이어의 움직임을 막아준다.
2. Sprite Renderer의 Sorting Layer를 Items로 변경하고 프리펩화 한다.



○ 게임 세팅

1. BoardManager 와 GameManager 스크립트를 작성
   - BoardManager : 플레이어가 새 게임을 시작할 때마다 랜덤으로 구성된 레벨을 생성(현재 레벨 단계에서)
   - GameManager : BoardManager를 게임에서 쓸 수 있게 해줌. 레벨을 로드하거나 플레리어 스코어를 관리
     -> 싱글턴 스크립트 추가 (싱글턴 : 게임 상에서 언제나 단 하나의 인스턴스만 존재할 수 있는 오브젝트, GameManager은 레벨 로드나 플레이어 스코어를 관리하기 때문에 하나 이상 있을 필요가 없으므로 코드로 확실하게 막아준다.)

2. GameManager EmptyObject를 만들고 BoardManager과 GameManager 스크립트를 넣고 변수 할당
   -> 인스펙터를 잠금 -> 프리펩들을 선택해 배열에 드래고 -> 세팅이 완료되면 인스펙터 잠금 해제



3. Loader 스크립트를 메인카메라에 더하고, GameManager 프리팹을 드래그하여 Loader에 넣어줌
   - Loader : 게임매니저가 인스턴스화 되었는지 체크해서 그렇지 않다면 프리팹으로부터 하나 인스턴스화 함

```
// Loader //
using UnityEngine;
using System.Collections;

public class Loader : MonoBehaviour {


    public GameObject gameManager;
        // Use this for initialization
        void Awake () {
        if (GameManager.instance == null) // GameManager가 null이면, 게임 매니저 프리펩을 인스턴스화 함
            Instantiate(gameManager);
        }
} // 완료되면 하이라카에서 게임매니저를 지워도 됨.
```

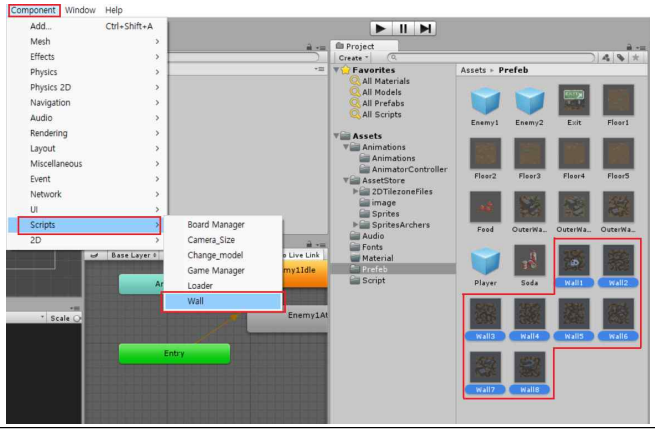
```
// GameManager //
using UnityEngine;
using System.Collections;
using System.Collections.Generic; // 적을 계속 추적하기 위해 사용할 리스트 자료 구조를 사용할 수 있게함

public class GameManager : MonoBehaviour {
    public float turnDelay = .1f; // 턴 사이에 게임이 얼마동안 대기할지 나타냄
    public static GameManager instance = null;
      // 클래스 바깥에서도 접근가능하며, static은 변수가 인스턴스가 아니라 클래스 그 자체에 속해있다는 것
         => 전역 변수, 어떤 스크립트에서도 접근가능
    public BoardManager boardScript;
    public int playerFoodPoints = 100;
    [HideInInspector] // 변수가 public이지만 에디터에서는 숨김
    public bool playersTurn = true;

    private int level = 3; // 적이 2레벨부터 등장하기 때문에 3
    private List<Enemy> enemies; // 적 리스트를 적들의 위치를 계속 추적하고 움직이도록 명령을 전달하기 위함
    private bool enemiesMoving;

        // Use this for initialization
    void Awake () { // Start()를 Awake()로 바꿈
        if (instance == null)
            instance = this; // 자기 자신의 위치 반환
        else if (instance != this)  // null도 this도 아닐 경우
            Destroy(gameObject); // 실수로 2개가 안생기게 방지
        DontDestroyOnLoad(gameObject); // 새로운 신을 로드할 때 점수가 사라지지 않게 함
        enemies = new List<Enemy>();
        boardScript = GetComponent<BoardManager>();
          // 콜바이레퍼런스 : 값 복사가 아닌 실제 오브젝트 자체를 들고 옴.
             보드매니저 스크립트로 컴포넌트들을 레퍼런스로 들고와 저장
        InitGame();
    }

 /* boardScript의 SetupScene 함수를 불러옴 */
    void InitGame () {
        enemies.Clear(); // 게임이 시작될 때 적 리스트를 초기화함, 이전 레벨의 적들을 전부 정리함
        boardScript.SetupScene(level);
    }

 /* 게임을 비활성화 함 */
    public void GameOver()
    {
        enabled = false;
    }

    void Update()
    {
        if (playersTurn || enemiesMoving) // playerTurn 혹은 적이 이미 이동 중이라는 뜻의 enemiesMoving이 참인지 체크함
            return; // 참이면 아래 코드를 실행하지 않음

        StartCoroutine(MoveEnemies()); // 둘다 거짓이면 실행
    }

 /* 적들이 자신을 게임 매니저에 등록하도록해서 게임 매니저가 적들이 움직이도록 명령할 수 있게 함 */
    public void AddEnemyToList(Enemy script)
    {
        enemies.Add(script);
    }

 /* 연속적으로 한 번에 하나씩 적을 옮기는데 사용 */
    IEnumerator MoveEnemies()
    {
        enemiesMoving = true;
        yield return new WaitForSeconds(turnDelay); // turnDelay(0.1초) 만큼 기다림
        if (enemies.Count == 0) // 적들이 아무도 없는지 체크, 첫 레벨이라는 뜻
            yield return new WaitForSeconds(turnDelay); // 대기하는 적이 없지만 일단 플레이어가 기다리게 함
        }

        for (int i = 0; i < enemies.Count; i++) // 적 리스트 만큼 루프
        {
            enemies[i].MoveEnemy(); // MoveEnemy함수를 호출해서 적들이 움직이도록 명령함
            yield return new WaitForSeconds(enemies[i].moveTime);
              // 다음 적을 호출하기 전에 yield 키워드와 적의 moveTime 변수를 입력하여 기다림
        }

        playersTurn = true;
        enemiesMoving = false;
    }

}
```

```
// BoardManager 1 //
using UnityEngine;
using System.Collections;
using System; // 직렬화 (Serializable) 속성들을 사용할 수 있다.
using System.Collections.Generic;  // List를 사용할 수 있다.
using Random = UnityEngine.Random; // System과 Unity Engine에 모두 랜덤이 있기 때문

public class BoardManager : MonoBehaviour {
    [Serializable]   // Count라는 Serializable(직렬화) public class 선언
    public class Count
    {
        public int minimum;
        public int maximum;

        public Count(int min, int max)  // 값 할당을 위한 생성자
        {
            minimum = min;
            maximum = max;
        }
    }
    public int columns = 8;      // 열을 위한 정수 (우리가 원하는 공간을 그림(크기조정))
    public int rows = 8;         // 행을 위한 정수
    public Count wallCount = new Count(5, 9); // Count를 사용해 레벨마다 얼마나 많은 벽을 랜덤하게 생성할지 범위를 특정
    public Count foodCount = new Count(1, 5);
    public GameObject exit; // 몇 변수를 선언하여 우리가 소환할 프리펩들을 잡아두게 해서 게임보드를 꾸미게함
    public GameObject[] floorTiles;
    public GameObject[] wallTiles;
    public GameObject[] foodTiles;
    public GameObject[] enemyTiles;     // 인스펙터에서 선택하게 될 여러 프리팹들로 각각의 배열을 채움
    public GameObject[] outerWallTiles;

    private Transform boardHolder;  // Hierarchy를 깨끗이 하기위해 사용(죄다 boardHolder 자식으로 집어넣음)
    private List<Vector3> gridPositions = new List<Vector3>();
      // 3차원 배열리스트, 게임판 위에 가능한 모든 다른 위치들을 추적하기 위해 사용하며,
      // 해당 장소에 있는지 없는지 추적하는데 사용

/* 리스트 초기화 */
    void InitializeList()
    {
        gridPositions.Clear();  // 리스트 내부의 clear 함수를 써서, 모든 리스트된 gridPosition을 초기화함
        for (int x = 1; x < columns - 1; x++)
        { // 게임상에서 벽이나 적, 아이템들이 있을 수 있는 가능한 모든 위치를 만듬
            for (int y = 1; y < rows - 1; y++)  // -1한 이유는 가장자리 때문
            {
                gridPositions.Add(new Vector3(x, y, 0f));  // 벡터 3를 격자형 위치 리스트에 더함
            }
        }
    }
```

```
// BoardManager 2 //
 /* void를 리턴하는 private 함수인 BoardSetup()선언, 바깥 벽과 게임 보드의 바닥을 짓기 위해 사용 */
    void BoardSetup()
    {
        boardHolder = new GameObject("Board").transform;
         // boardHolder를 Boad라는 새로운 게임 오브젝트의 Transform과 동일하게 하고 시작
        for (int x = -1; x < columns + 1; x++) // -1과 +1은 바닥과 바깥벽을 구분하기 위함
        {
            for (int y = -1; y < rows + 1; y++)
            {
                GameObject toInstantiate = floorTiles[Random.Range(0, floorTiles.Length)];
                  // 타일 게임 오브젝트형 변수를 선언한 것. 0에서 floorTile 길이 사이에서 랜덤하게 골라온
                     floorTiles 오브젝트 배열의 인덱스와 같은 값을 갖게함.
                  // 길이를 미리 지정할 필요가 없어짐, length 값을 가지고 배열에 있는 값 중 하나 선택
                if (x == -1 || x == columns || y == -1 || y == rows)  // 인스턴스화할 바깥 벽 타일 선택
                    toInstantiate = outerWallTiles[Random.Range(0, outerWallTiles.Length)];
                        // 바깥 벽 타일의 배열 중 랜덤하게 고른 타일로 인스턴스화를 준비
                GameObject instance = Instantiate(toInstantiate, new Vector3(x, y, 0f), Quaternion.identity) as GameObject;
                  // instance라 불리는 게임 오브젝트 타입을 선언하고, 인스턴스화 하려는 오브젝트를 할당,
                     따라서 Instatiate함수를 불러오고, 우리가 고른 프리펩인 toInstantiate를 넣어 현재 루프의
                     x와 y 위치 값이 있는 new Vector3를 더함
                instance.transform.SetParent(boardHolder);
                  // 새로 생성된 인스턴스의 부모 오브젝트를 boardHolder로 해줌
            }
        }
    }

 /* Vector3를 리턴하는 새로운 함수 */
    Vector3 RandomPosition()
    {
        int randomIndex = Random.Range(0, gridPositions.Count);
          // randomIndex를 선언하고, Range 내에 0부터 gridPositions 리스트의 위치 값 개수만큼 랜덤 값을 설정
        Vector3 randomPosition = gridPositions[randomIndex];
          // gridPositions 리스트에서 랜덤하게 선택된 인덱스에 있는 격자 위치 값과 동일하게
        gridPositions.RemoveAt(randomIndex);
          // 같은 장소에 두 개 이상의 오브젝트를 만드는 것을 방지, 사용한 격자 위치를 리스트에서 제거
        return randomPosition;
    }

 /* 선택한 장소에 실제로 타일을 소환 */
    void LayoutObjectAtRandom(GameObject[] tileArray, int minimum, int maximum)
    {                       // 세가지 파라미트 [게임 오브젝트 배열 tileArray, minimum, maximum]
        int objectCount = Random.Range(minimum, maximum + 1);
          // 주어진 오브젝트를 얼마나 스폰할지 조정 ex)한 레벨 벽 개수
        for (int i = 0; i < objectCount; i++)
        {
            Vector3 randomPosition = RandomPosition();  // 랜덤 위치를 선택
            GameObject tileChoice = tileArray[Random.Range(0, tileArray.Length)];
              // 0부터 tileArray.length까지 랜덤 값을 생성해 넣음 (소환할 랜덤타일 선택)

            Instantiate(tileChoice, randomPosition, Quaternion.identity);
              // 우리가 선택한 랜덤위치에 인스턴스화 함
        }
    }

 // void Start()와 void Update()를 지우고 public void SetupScene 선언
 /* 유일한 public 함수, 실제로 게임보드가 만들어질 때 게임매니저에 의해 호출*/
    public void SetupScene(int level) {
        BoardSetup();  // 신을 구성하기 위해 호출
        InitializeList();
        LayoutObjectAtRandom(wallTiles, wallCount.minimum, wallCount.maximum);
        LayoutObjectAtRandom(foodTiles, foodCount.minimum, foodCount.maximum);
        int enemyCount = (int)Mathf.Log(level, 2f);
          // 레벨에 따라 적 생성, 로그함수를 사용 ex) 1스테이지 : 1마리, 4스테이지 : 2마리, 8스테이지 : 3마리
        LayoutObjectAtRandom(enemyTiles, enemyCount, enemyCount); // min, max 값이 같음
        Instantiate(exit, new Vector3(columns - 1, rows - 1, 0f), Quaternion.identity);
          // 출구는 언제나 같은 곳 => 그냥 Instantiate호출
    }
}
```

○ 조작 세팅 (Moving)

1. MovingObject 스크립트를 작성
   - public abstact 클래스로 만드는데, 기능을 완성하지 않아도 되고, 해당 클래스는 반드시 파생클래스로 상입되야 함.
2. Wall 스크립트 작성
   - 플레이어가 벽에 막혔을 때 벽을 부수게 해줌
3. 프리팹의 Wall1~8까지 드래그 후 Component 탭의 Scripts에서 Wall을 적용하고, Sprite를 넣어줌



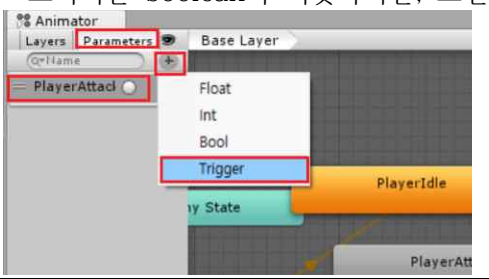

```
// Wall 스크립트 //
using UnityEngine;
using System.Collections;

public class Wall : MonoBehaviour {

    public Sprite dmgSprite; // 플레이어가 벽을 한 번 때렸을 때 보여줄 스프라이트
    public int hp = 4; // 벽의 체력

    private SpriteRenderer spriteRenderer;

        void Awake () {
        spriteRenderer = GetComponent<SpriteRenderer>();
        }

    public void DamageWall(int loss)
    {
        spriteRenderer.sprite = dmgSprite;
         // spriteRenderer의 스프라이트를 dmgSprite 것으로 바꾸어 줌
         // 플레이어가 성공적으로 벽을 공격할 때 시각적인 변화를 줌)
        hp -= loss;
        if (hp <= 0)
            gameObject.SetActive(false);
    }
}
```

```
// MovingObject 스크립트 -1 //
using UnityEngine;
using System.Collections;
/* 제너릭을 사용하는 이유는 플레이어와 적이 MoveObject를 상속하기 때문에 플레이어는 벽과 상호작용해야 하고,
   적은 플레이어와 상호작용할 수 있어야 함. 이는 나중에 상호작용할 hitComponent의 종류를 알 수 없다는 뜻.
   일반형을 사용함으로서 당장 이들의 레퍼런스를 OnCantMove로 가져와 입력할 수 있고, 이를 상속한 클래스들의
   각각의 구현에 따라 동작하게 할 수 있음. */
public abstract class MovingObject : MonoBehaviour {
    public float moveTime = 0.1f; // 수 초 동안 오브젝트를 움직이게 할 시간 단위
    public LayerMask blockingLayer; // 이동할 공간이 열려있고, 그 곳으로 이동하려 할 때 충돌이 일어났는지 체크

    private BoxCollider2D boxCollider;
    private Rigidbody2D rb2D; // 움직일 유닛의 rigidbody 2D 컴포넌트의 레퍼런스를 저장
    private float inverseMoveTime; // 움직임을 더 효율적으로 계산

        // Use this for initialization
        protected virtual void Start () { // protected virtual 함수는 자식 클래스가 덮어써서 재정의할 수 있다.(오버라이딩)
        boxCollider = GetComponent<BoxCollider2D>();
        rb2D = GetComponent<Rigidbody2D>();
        inverseMoveTime = 1f / moveTime; // moveTime의 역수를 저장함으로서, 나누기 대신 효율적인 곱하기를 쓸 수 있다.
        }

    protected bool Move (int xDir, int yDir, out RaycastHit2D hit) // out 키워드는 입력을 레퍼런스로 받게 함
    { //Move 함수가 두 개 이상의 값을 리턴하기 위해 쓰임, 함수에 의해 리턴되는 bool 값이 있고,
      //hit라는 RayCastHit2D 또한 리턴
        Vector2 start = transform.position; // 현재 오브젝트의 위치(transform.position)를 저장하기 위해 사용,
                                            // 2차원 벡터로 형변환
        Vector2 end = start + new Vector2(xDir, yDir);
          // Move 함수를 부를 때, 입력받은 방향 값들을 기반으로 끝나는 위치를 계산하는데 사용

        boxCollider.enabled = false; // Ray를 사용할 때 자기 자신의 충돌체에 부딪치지 않게 하기 위해 boxCollider를 해제
        hit = Physics2D.Linecast(start, end, blockingLayer); // BlockingLayer와의 충돌을 검사
        boxCollider.enabled = true; // 다시 boxCollider를 활성화

        if (hit.transform == null)  // 뭔가 부딪쳤는지 (hit.transform이 null인지) 체크
        {
            StartCoroutine(SmoothMovement(end));
             // 라인으로 검사한 공간이 열려있고, 그 곳으로 이동할 수 있음
             // end를 입력받는 SmoothMovement 코루틴을 시작함
            return true; // 이동할 수 있기 때문에 true를 리턴
        }

        return false; // 이동할 수 없기 때문에 false를 리턴
    }
```

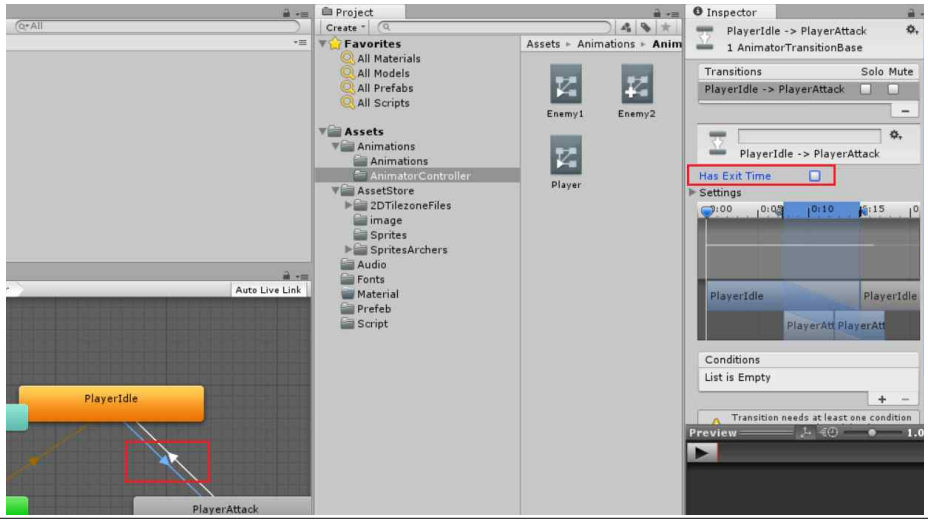
```
// MovingObject 스크립트 -2 //
 /- 유닛들을 한 공간에서 다른 곳으로 옮기는데 쓰임 */
    protected IEnumerator SmoothMovement (Vector3 end)
    {
        float sqrRemainingDistance = (transform.position - end).sqrMagnitude;
         // 이동할 거리 계산 / end와 현재 위치의 차이 벡터에 sqrMagintude로 거리 계산
         // (* Magintude - 벡터 길이, sqrMagintude : 벡터 길이 제곱)

        while (sqrRemainingDistance > float.Epsilon) { // 남은 거리의 제곱이 float.Epsilon보다 큰지 확인
            Vector3 newPosition = Vector3.MoveTowards(rb2D.position, end, inverseMoveTime * Time.deltaTime);
            // 루프에서 moveTime에 기반해서 적절이 end에 가까운 새로운 위치를 찾음
            // 현재 포인트를 직선상의 목표 포인트로 이동시키는 Vector3.MoveTowards를 사용
            // newPosition : 이동시킬 포지션, rb2D.position : 처음 값, end : 목적지
            rb2D.MovePosition(newPosition);
              // 새 지점으로 이동시키기 위해 rigid body 2D의 MovePosition 함수 사용
            sqrRemainingDistance = (transform.position - end).sqrMagnitude; // 남은 거리를 계산
            yield return null; // 루프를 갱신하기 전에 다음 프레임을 기다림
        }
    }

 /* void를 리턴하며, X방향과 Y방향을 위한 두 개의 정수 값을 입력 받음 */
    protected virtual void AttemptMove<T>(int xDir, int yDir)
        where T : Component // 일반형 입력 T는 막혔을 때, 유닛이 반응할 컴포넌트 타입을 가리키기 위해 사용
    { // 적에 적용된 경우 상대 : 플레이어, 플레이어일 경우 상대 : 벽들이기 때문에 플레이어가 벽을 공격하고 파괴할 수 있음
        RaycastHit2D hit;
        bool canMove = Move(xDir, yDir, out hit); // 이동하는데 성공하면 canMove가 true, 실패하면 false가 됨
         // hit이 Move의 out으로 들어갔기 때문에 Move에서 부딪친 transform이 null인지 확인할 수 있음
        if (hit.transform == null)
            return;  // Move에서 라인 캐스트가 다른 것과 부딪치지 않았다면 리턴하고 코드를 실행하지 않음
        T hitComponent = hit.transform.GetComponent<T>();
          // 부딪쳤다면, 충돌한 오브젝트의 컴포넌트의 레퍼런스를 T타입의 컴포넌트에 할당함
          // 움직이던 오브젝트가 막혔고, 상호작용할 수 있는 오브젝트와 충돌했음을 뜻함
        if (!canMove && hitComponent != null)
            OnCantMove(hitComponent);
    }

    protected abstract void OnCantMove<T>(T component)
      // OnCantMove는 추상화 함수이기 때문에 괄호를 사용하지 않음
      // 여기 추상문은 사용할 것들이 현재 존재하지 않거나, 불완전하게 만들어졌음을 의미(* 상속한 자식 클래스에서 완성하면 됨)
        where T : Component; // 일반형 입력 T를 T형의 component라는 변수로서 받아옴
}
```

○ 플레이어 애니메이션 세팅 (Animator)

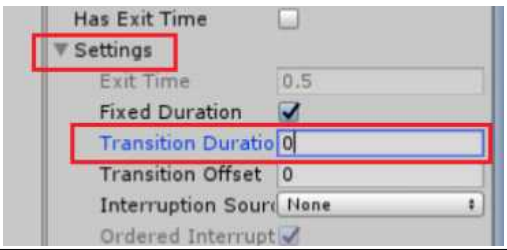
1. Player Animation Controller를 실행 후 Parameters의 + 버튼을 눌러 트리거(Trigger) 타입의 파라미터를 더해주고 각각 PlayerAttack과 PlayerHit으로 설정한다.
   - 트리거는 boolean과 비슷하지만, 트랜지션에 사용된 후 즉시 리셋된다. (true가 되었다가 즉시 false)

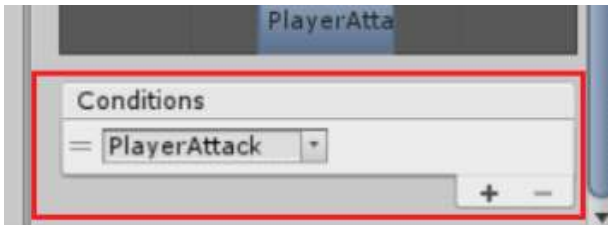


2. 상태를 우클릭 후 Make Transition을 선택하고 각자에 맞는 트랜지션을 설정 후 화살표를 눌러 Inspector 에 띄운 후 Has Exit Time의 설정을 해제한다.
   - Has Exit Time : 애니메이션이 다 끝나거나 특정 지점까지 재생되는 것을 기다림

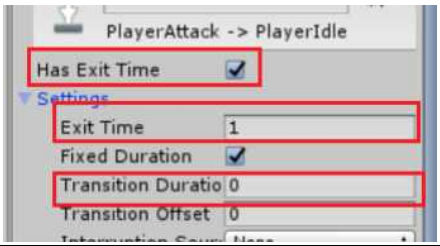


3. Idle-> Attack의 Transition Duration(트랜지션 지속시간)을 0으로 설정하고, 하단의 조건 리스트를 고른다.
   - 3D 애니메이션에서는 블렌드(혼합)을 사용하지만 2D에서는 불가능하다.
   - 스크립트에서 PlayerAttack의 트리거가 발동하면 PlayerAttack으로 트랜지션한다.





4. Idle상태로 돌아가는 트랜지션에서는 Has Exit Time을 활성화하고 Exit Time을 1로 설정하고, Transition Duration(트랜지션 지속시간)을 0으로 설정한다.
   - Exit Time에 도달했을 때 트랜지션이 적용된다.



○ 조작 세팅 - Player

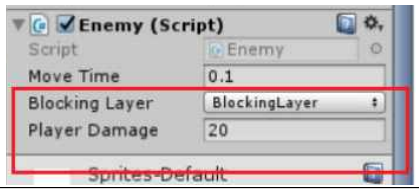
1. Player 스크립트를 작성 후 Player 컴포넌트에 더해준다.

```
// Player 스크립트 - 1 //
using UnityEngine;
using System.Collections;

public class Player : MovingObject {

    public int wallDamage = 1; // 플레이어가 벽을 부술 때 적용되는 데미지
    public int pointsPerFood = 10; // 스코어에 더해질 변수
    public int ponitsPerSoda = 20;
    public float restartLevelDelay = 1;
    private int food; // 레벨을 바꾸면서 스코어를 다시 게임매니저로 입력해 넣기 전에, 해당 레벨동안의 플레이어 스코어를 저장

    private Animator animator; // 애니메이터 컴포넌트의 레퍼런스를 가져오기 위해 사용
        // Use this for initialization
        protected override void Start () { // MovingObject에 있는 Start와는 다르게, Player Start를 구현할 것이기 때문
        animator = GetComponent<Animator>();

        food = GameManager.instance.playerFoodPoints; // 레벨이 바뀔 때 게임매니저로 다시 저장

        base.Start(); // base(부모의 클래스)의 Start를 불러옴
        }

    private void OnDisable() // OnDisable은 유니티 API에 속한 함수, 게임 오브젝트가 비활성화 되는 순간 호출
    {                        // 레벨이 변환될 때 게임매니저에 food 값을 저장하는데 사용
        GameManager.instance.playerFoodPoints = food;
    }
        // Update is called once per frame
        void Update () {
        if (!GameManager.instance.playersTurn) return;
          // 만약 플레이어의 턴이 아니라면 return을 사용하여 이하 코드들이 실행되지 않음

        int horizontal = 0;
        int vertical = 0; // 수평이나 수직으로 움직이는 방향을 1이나 -1로서 저장해서 사용함

        horizontal = (int)Input.GetAxisRaw("Horizontal");
        vertical = (int)Input.GetAxisRaw("Vertical");

        if (horizontal != 0)
            vertical = 0; // 플레이어가 대각선으로 움직이지 않게하기 위함

        if (horizontal != 0 || vertical != 0)
            AttemptMove<Wall>(horizontal, vertical); // 플레이어가 상호작용할 수 있는 벽에 대면할지도 모름
             // MovingObject를 작성할 때 AttemptMove 함수에 상호작용할 오브젝트가 될 일반형 입력 T를
             // 줬기 때문에 함수를 호출할 때 상호작용할 컴포넌트를 특정할 수 있음
        }

    protected override void AttemptMove<T>(int xDir, int yDir)
    {  // <T>는 움직이는 오브젝트가 마주칠 대상의 컴포넌트의 타입을 가리킴
        food--; // 플레이어가 움직일 때 마다 음식 점수를 1씩 잃음

        base.AttemptMove<T>(xDir, yDir);

        RaycastHit2D hit; // Move 함수에서 이루어진 라인캐스트 충돌 결과의 레퍼런스를 가져올 RayCastHit2D 타입의 hit 선언

        CheckIfGameOver();
           // 플레이어가 움직이면서 음식 점수를 잃었기 때문에 우리는 CheckIfGameOver를 호출하여 게임이 끝났는지 확인
        GameManager.instance.playersTurn = false; // 플레이어의 턴이 끝남
    }
```

```
// Player 스크립트 - 2 //
 /* 플레이어가 게임의 코어 메카닉을 사용하게 해줌 */
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.tag == "Exit")
        {
            Invoke("Restart", restartLevelDelay); // 1초간 정지하고 레벨을 다시 시작함
            enabled = false;
        }
        else if (other.tag == "Food")
        {
            food += pointsPerFood;
            other.gameObject.SetActive(false);
        }
        else if (other.tag == "Soda")
        {
            food += pointsPerSoda;
            other.gameObject.SetActive(false);
        }
    }

 /* 플레이어가 이동하려는 공간에 벽이 있고, 이에 막히는 경우의 행동을 작성 */
    protected override void OnCantMove<T>(T component)
    {
        Wall hitWall = component as Wall;
        hitWall.DamageWall(wallDamage); // wallDamage : 얼마나 플레이어가 벽에 데미지를 줄지 알림
        animator.SetRrigger("PlayerAttack"); // 트리거 발동
    }

 /* 플레이어가 출구와 충돌했을 때 방 재시작 및 다음 레벨로 넘어감 */
    private void Restart()
    {
        Application.LoadLevel(Application.loadedLevel);
    }

 /* 적이 플레이어를 공격할 때 호출됨 */
    public void LoseFood(int loss)
    {
        animator.SetTrigger("PlayerHit");
        food -= loss;
        CheckIfGameOver();
    }

    private void CheckIfGameOver()
    {
        if (food <= 0)
            GameManager.instance.GameOver();
    }
}
```

○ 조작 세팅 - Enemy

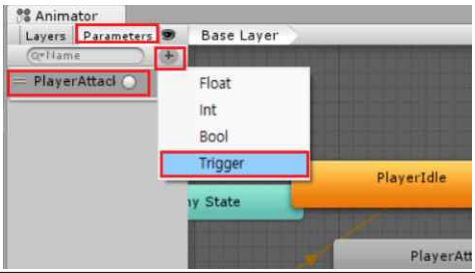
1. Enemy 스크립트를 작성 후 Enemy 컴포넌트에 더해준다.
2. Enemy 스크립트의 Blocking Layer를 설정하고, Player Damage를 각각 10, 20으로 설정한다.



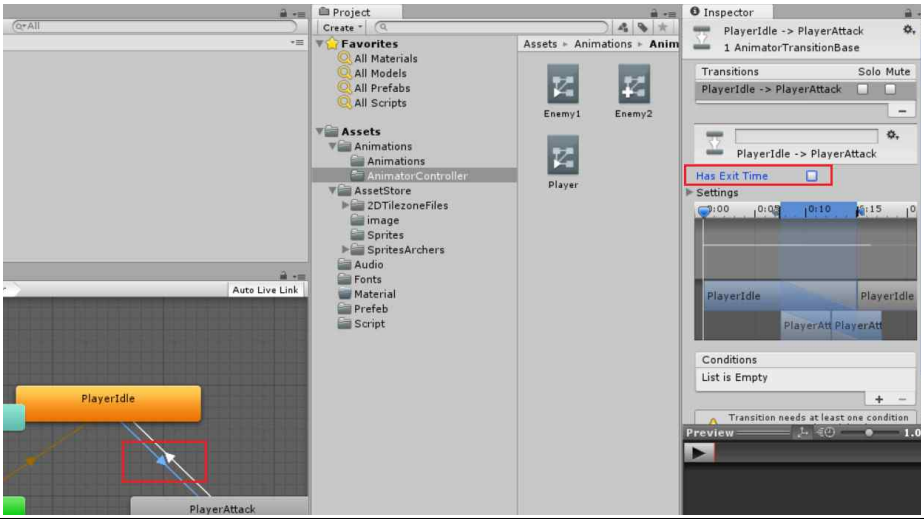

```
// Enemy 스크립트 //
using UnityEngine;
using System.Collections;

public class Enemy : MovingObject{

    public int playerDamage; // 적이 플레이어를 공격할 때 뺄 음식 포인트

    private Animator animator;
    private Transform target; // 플레이어의 위치를 저장하고, 적이 어디로 향할지 알려줌
    private bool skipMove;

    protected override void Start () {
        GameManager.instance.AddEnemyToList(this);
          // Enemy 스크립트가 자기 스스로를 게임매니저의 적 리스트에 더해줌
          // 이 코드로 게임매니저는 Enemy에서 public으로 선언한 MoveEnemy 함수를 호출할 수 있음
         animator = GetComponent<Animator>(); // 애니메이터 컴포넌트의 레퍼런스를 animator에 가져와 저장
        target = GameObject.FindGameObjectWithTag("Player").transform; // target변수에 플레이어 Transform을 저장
        base.Start(); // 부모 클래스 Start를 불러옴
    }

    protected override void  AttemptMove<T>(int xDir, int yDir)
    { // 적이 상호작용할 것으로 예상되는 플레이어를 입력
        if (skipMove){ // 참이면 적의 턴을 스킵
            skipMove = false;
            return;
        }
            base.AttemptMove<T>(xDir, yDir); // 플레이어가 될 일반형 입력 T를 입력하고, 이동할 x와 y방향을 함께 입력
        skipMove = true;
    }

 /* 적들이 움직이려할 때 게임 매니저에 의해 호출 */
    public void MoveEnemy(){
        int xDir = 0;
        int yDir = 0;

        if (Mathf.Abs (target.position.x - transform.position.x) < float.Epsilon)
         // 0인 앱실론(극한)보다 작은지 체크, 즉, x좌표가 대충 같은지 체크하는 것이고, 적과 플레이어가 같은 열에 속한다는 의미
            yDir = target.position.y > transform.position.y ? 1 : -1;
             // 같은 열에 있으면 target y좌표가 transform y좌표 보다 큰지 체크, 크면 위로이동, 작으면 아래로 이동
        else
            xDir = target.position.x > transform.position.x ? 1 : -1;
                // x좌표 값을 비교하여 x방향 값을 정함
        AttemptMove <Player> (xDir, yDir);
    }

 /* 플레이어가 점거중인 공간으로 적이 이동하려고 시도할 때 호출 */
    protected override void OnCantMove <T> (T component){
        // 추상 함수로 선언한 MovingObject의 OnCantMove를 재정의함
        Player hitPlayer = component as Player; // 입력한 컴포넌트를 Player로 형변환해서 = 연산

        animator.SetTrigger("EnemyAttack");

        hitPlayer.LoseFood(playerDamage);
         // 플레이어의 LoseFood를 호출하고, 적의 공격으로 인해 잃어버릴 음식 점수가 될 playerDamage를 입력
    }
}
```

○ 적 애니메이션 세팅 (Animator)

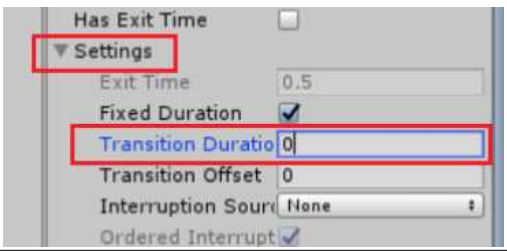
1. Enemy1 Animation Controller를 실행 후 Parameters의 + 버튼을 눌러 트리거(Trigger) 타입의 파라미터를 더해주고 각각 EnemyAttack으로 설정한다.

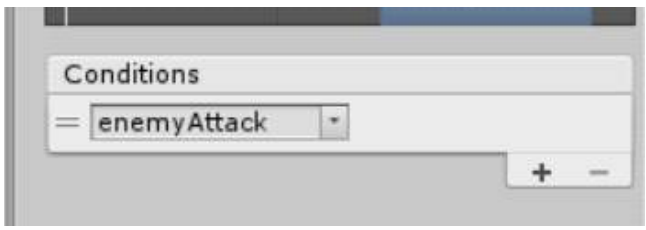


2. 상태를 우클릭 후 Make Transition을 선택하고 각자에 맞는 트랜지션을 설정 후 화살표를 눌러 Inspector 에 띄운 후 Has Exit Time의 설정을 해제한다.
   - Has Exit Time : 애니메이션이 다 끝나거나 특정 지점까지 재생되는 것을 기다림

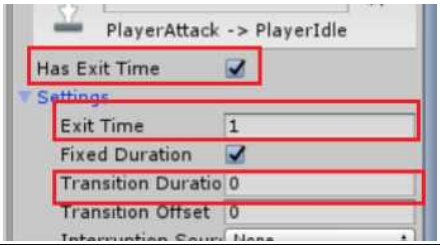


3. Idle-> Attack의 Transition Duration(트랜지션 지속시간)을 0으로 설정하고, 하단의 조건 리스트를 고른다.
   - 3D 애니메이션에서는 블렌드(혼합)을 사용하지만 2D에서는 불가능하다.
   - 스크립트에서 Enemy1Attack의 트리거가 발동하면 Enemy1Attack으로 트랜지션한다.



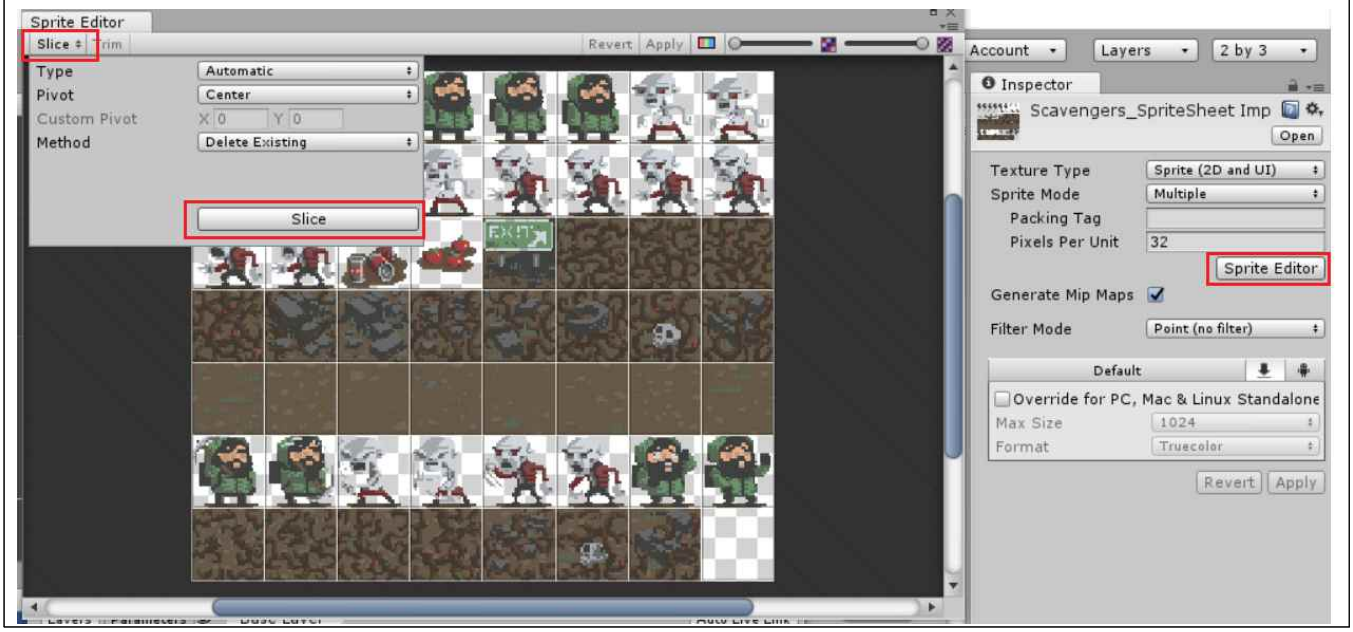


4. Idle상태로 돌아가는 트랜지션에서는 Has Exit Time을 활성화하고 Exit Time을 1로 설정하고, Transition Duration(트랜지션 지속시간)을 0으로 설정한다.
   - Exit Time에 도달했을 때 트랜지션이 적용된다.



○ 스프라이트 나누는 방법

Sprite Editor 클릭 후 Slice를 하여 중앙점을 지정한다.



실습