UNITY
-CAHPTER6-

SOUL SEEK

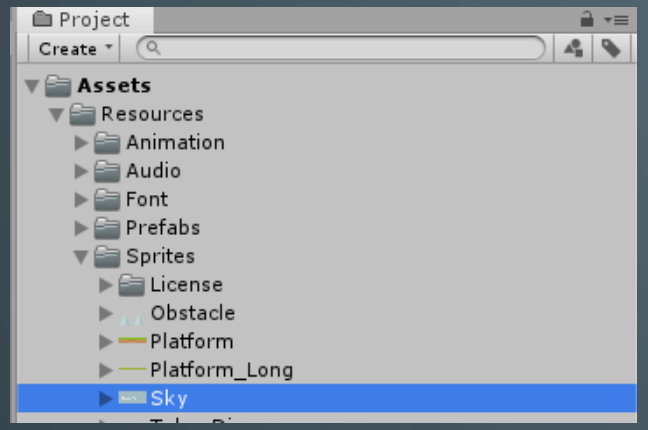
목차

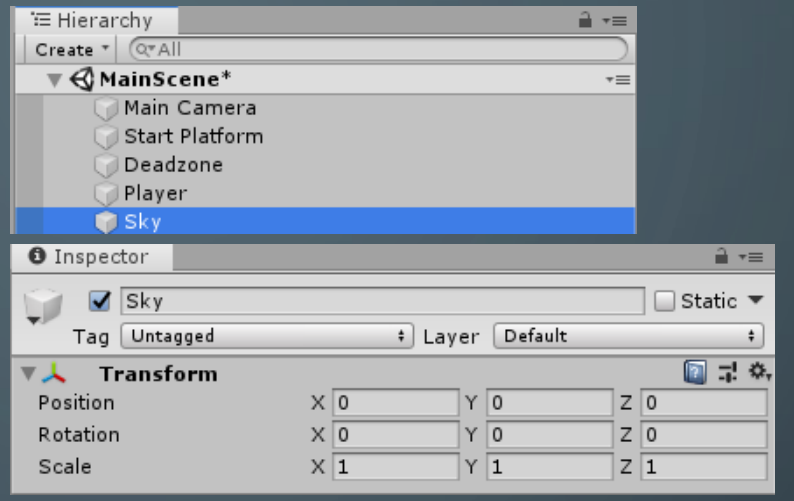
1. 배경스크롤
2. 해상도에 맞는 UI

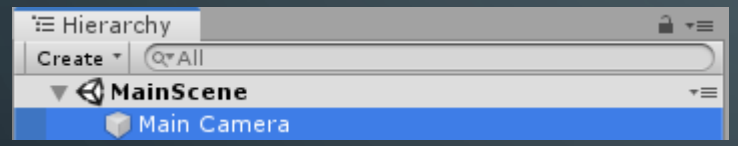
배경 스크롤

1. 준비하기

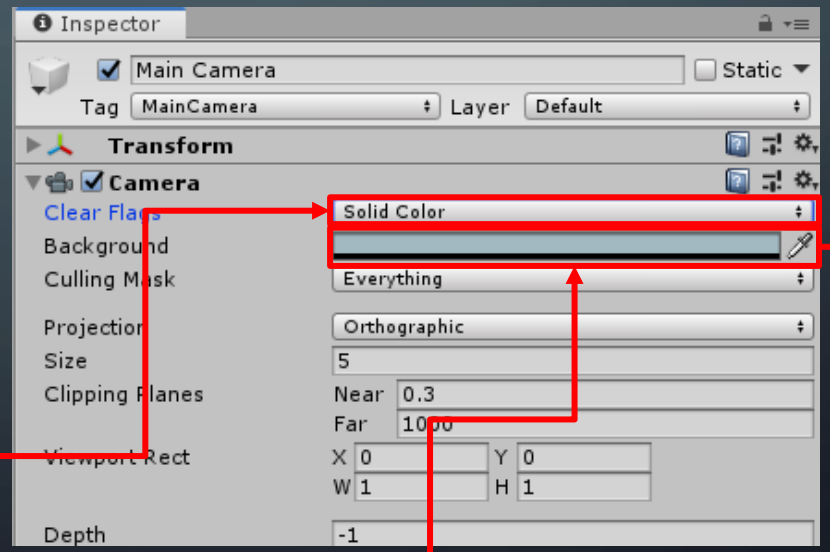
- **Scroll**할 배경을 **GameObject**로 추가



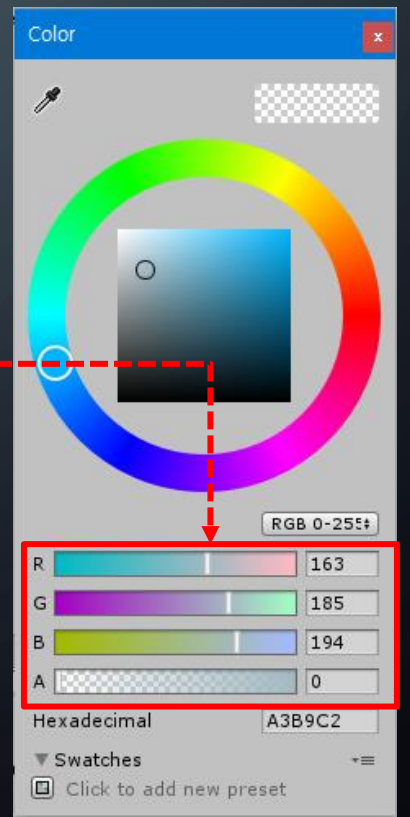




GameObject이외의 공간의 이질감을 없애기 위해 카메라 속성 변경



Clear Flags를 **Solid Color**로 변경

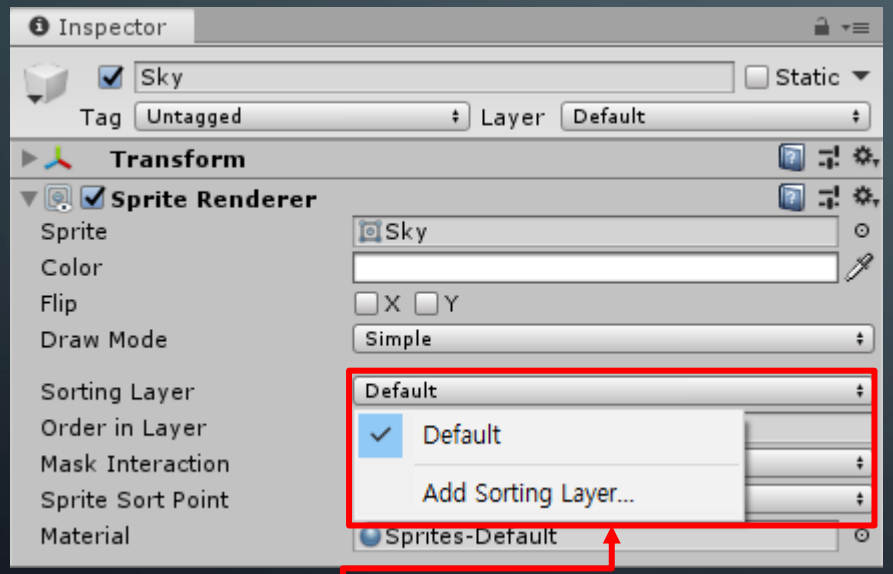


Background Color Field 클릭 > 컬러를 **(163, 185, 194)**로 변경

2. LAYER SORT

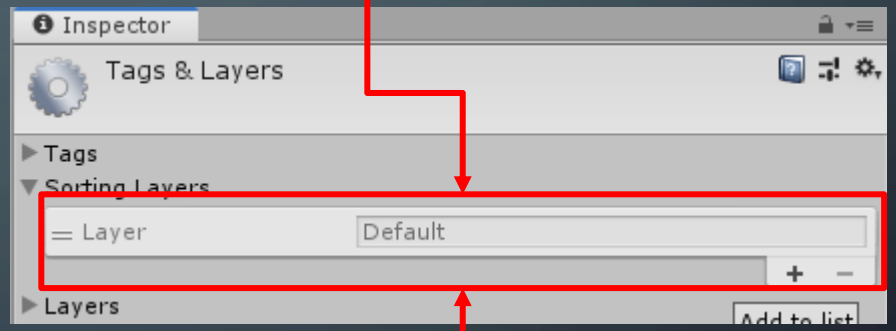
- 배경을 추가하면 추가한 배경에 의해 **Player**나 지형이 가려져 보일 때도 있다.
- **2D**에서는 **Z**축이 없다고 생각하기 때문에 **World**에 존재하는 **Object**의 앞쪽 뒤쪽 관계를 **Sort**로 관리해야 한다. → **UI**에서 많이 사용.
- **XXXRenderer**라는 **Component**에는 **Rendering Sort** 설정이 있다.
  → 좌표와 상관없이 **Rendering** 순서에 의한 앞뒤 표현을 하는 것이다.
  → 일일이 좌표를 계산할 필요가 없다.

Layer Sort를 설정하고 적용하자.

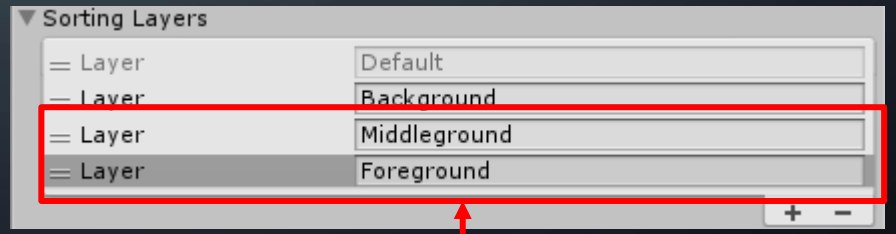


Sorting Layer의 **Default** 클릭 > **Add Sorting Layers** 클릭

Sorting Layers 리스트의 **+** 버튼 클릭



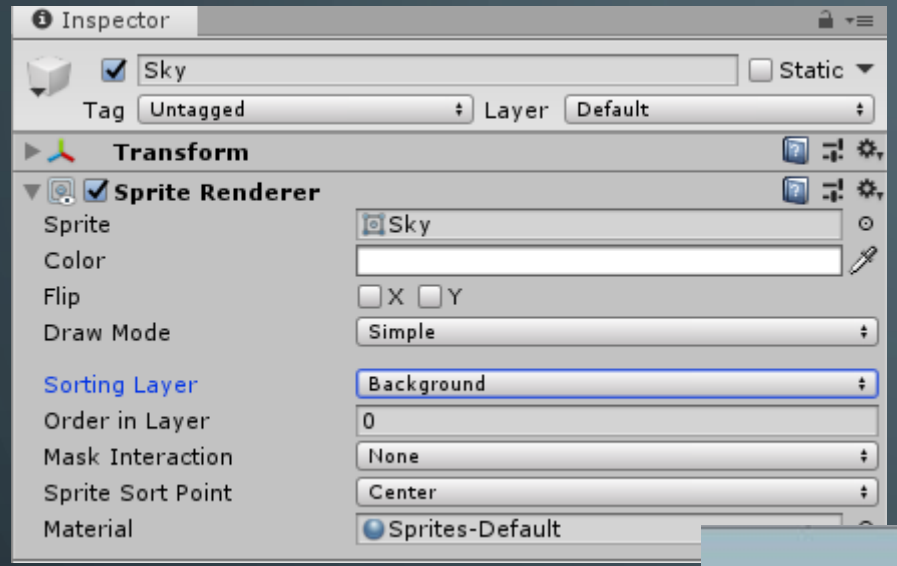
생성된 **Layers** 이름을 **Background**로 변경

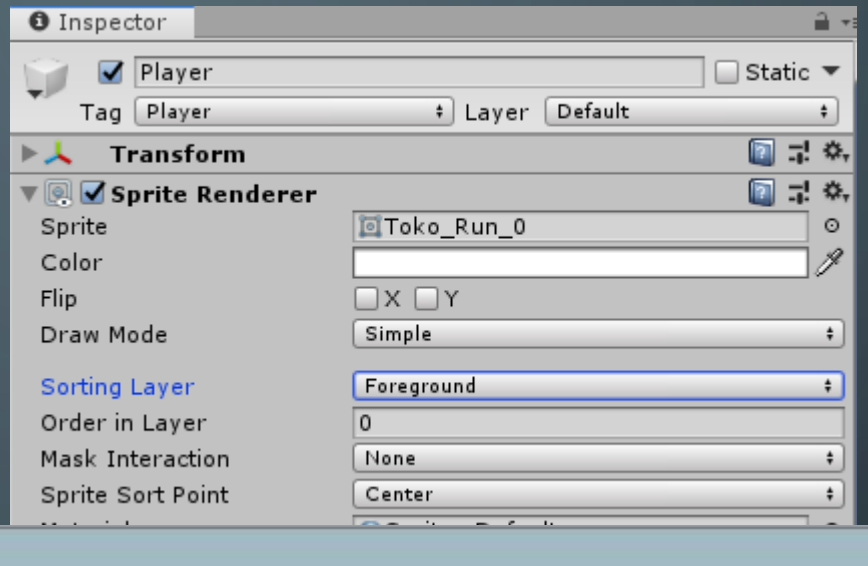


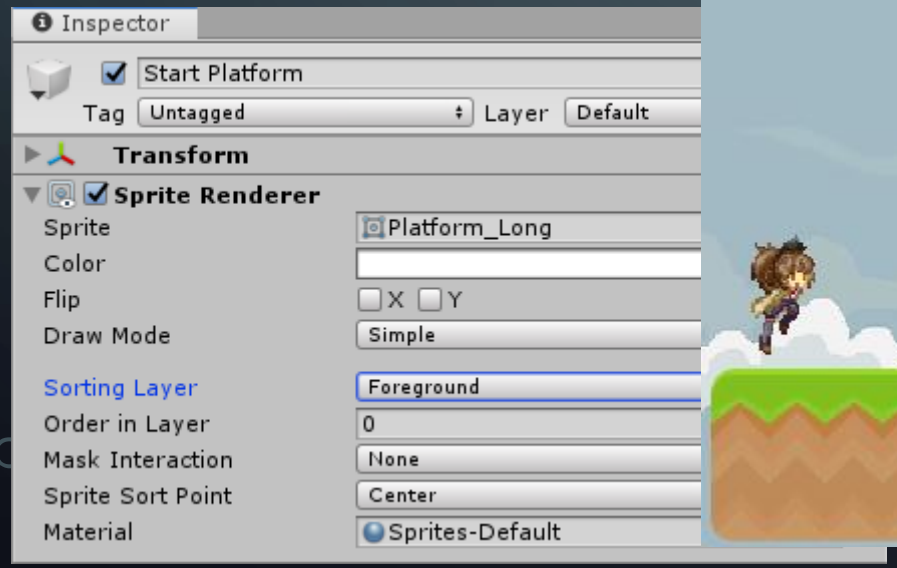
Middleground, Foreground도 추가한다.

2. LAYER SORT

- 추가한 **Layer**를 적용해서 실행해보자.







3. 배경 스크롤

- **Scrolling**할 **Object**에 **ScrollingObject Component**를 추가하자**.**
- →**Sky, StartPlatform**에 **ScrollingObject Component**를 추가

ScrollingObject Script를 살펴보자

```
// 게임 오브젝트를 계속 왼쪽으로 움직이는 스크립트
public class ScrollingObject : MonoBehaviour
{
    public float speed = 10f; // 이동 속도

    private void Update()
    {
        // 게임 오브젝트를 왼쪽으로 일정 속도로 평행 이동하는 처리
    }
}
```

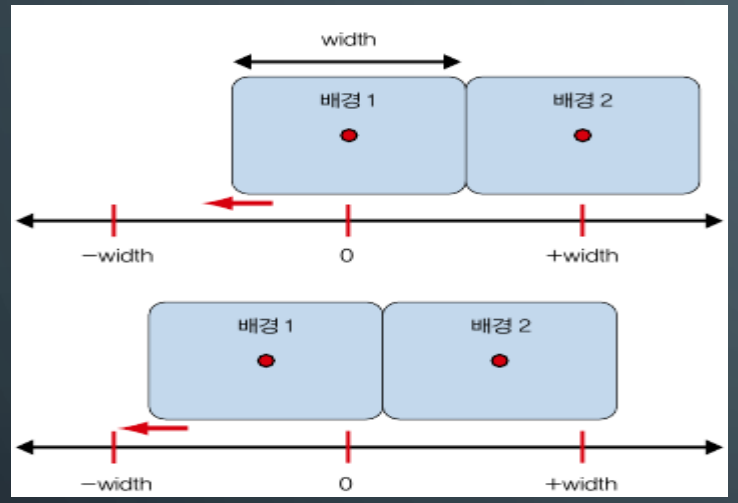
Component를 추가한 **GameObject**들의 **Transform**을 초당 속도와 방향으로 움직이게 하자**.**

```
private void Update()
{
    // 초당 speed의 속도로 왼쪽으로 평행이동
    transform.Translate(Vector3.left * speed * Time.deltaTime);
}
```

3. 배경 스크롤

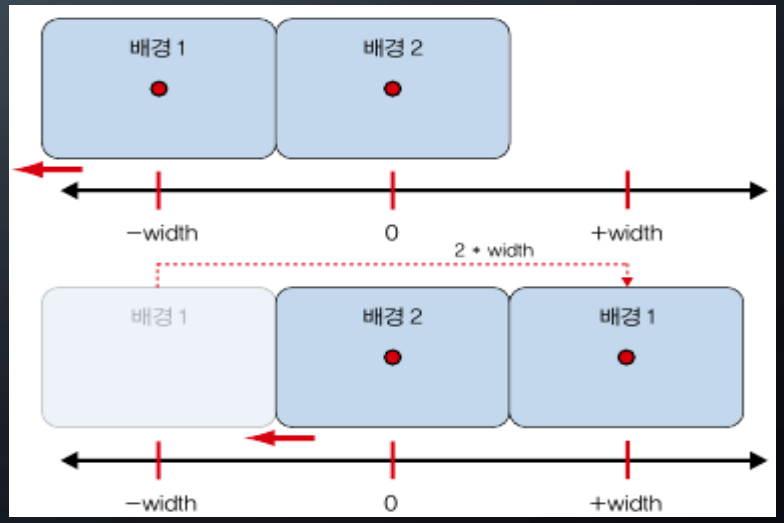
반복되는 배경을 만들려고 할 때 고려해야 할 사항

- 적어도 두 개의 배경이 있어야 끊기지 않고 반복되는 모습을 연출 할 수 있다.
- 사용한 배경의 **width**값을 기준으로 **width**값만큼 이동했다면 오른쪽 끝으로 이동하게 하는 방법을 사용한다.



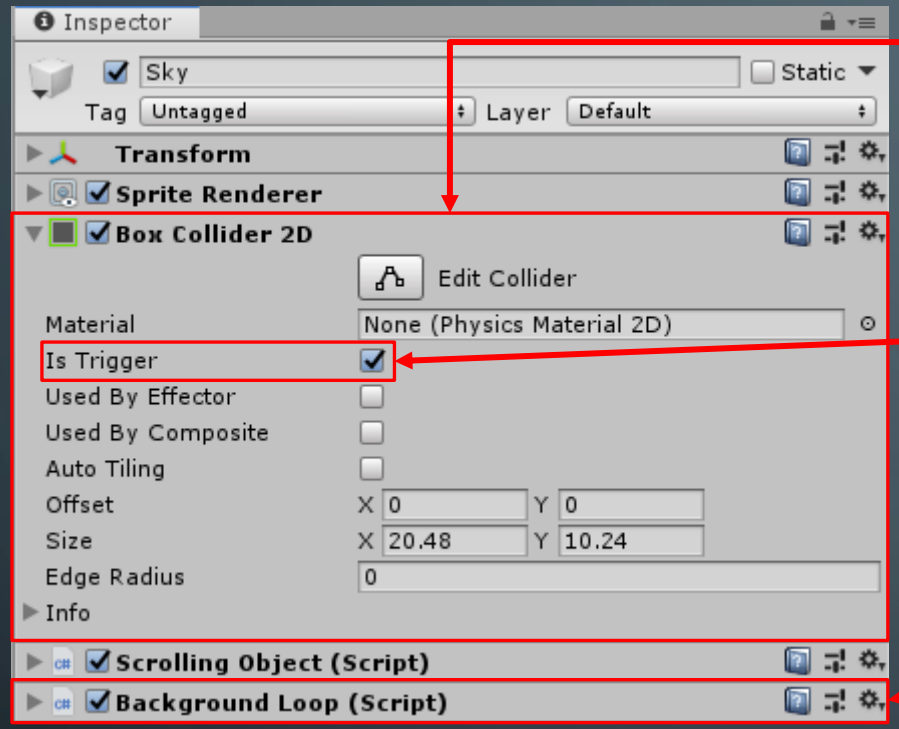
- **BackgroundLoop Script**는 **GameObject**의 **X**축 위치 **(transfo rm.position.x)**를 계속 체크한다. 만약, **GameObject**의 **X**축 위치값이 **–width** 이하라면 '너무 많이 왼쪽으로' 이동했다고 판단하고 현재 위치에 **2 * width**를 더 해서 되 돌려놓는다.

- **GameObject**의 **X**축 위치가 **–width**가 되는 순간 **+width**로 변경되어**(**오른쪽으로 순간이동하여**)** 배경 스크롤링이 무한 반복된다.
- **width**의 값을 구해오는 것이 문제인데 **Scrolling**할 **Object** 들에게 **Collider**를 추가해 두면 이 **Collider**는 **Object**의 크기를 체크할 수 있다.



3. 배경 스크롤

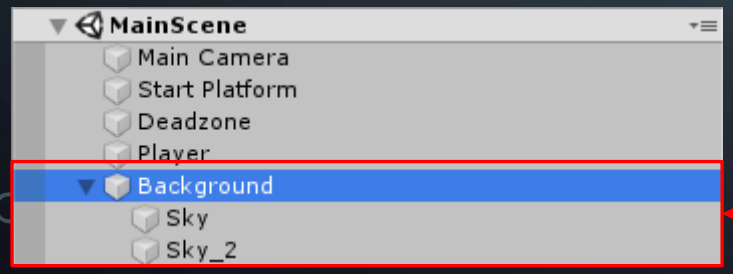
Scrolling할 **Object**들에게 **Collider**를 추가하고 **BackgroundLoop** 스크립트를 추가해서 **Scrolling**을 반복해 보자.

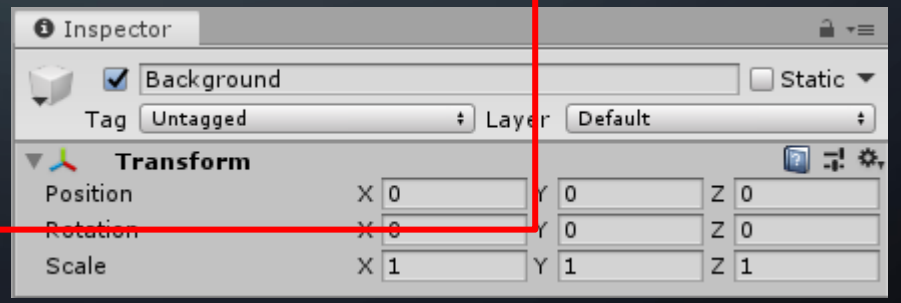


BoxCollider 2D Component를 추가한다.

물리작용을 받지 않고 충돌이벤트 체크용으로 **is Trigger**를 체크한다.

Scrolling을 반복적으로 **Loop**할 **Object**에 추가한 뒤 **Loop**할 **Object**를 한곳에 모아둔다.





3. 배경 스크롤

BackgroundLoop Component를 살펴보자**.**

```
// 왼쪽 끝으로 이동한 배경을 오른쪽 끝으로 재배치하는 스크립트
public class BackgroundLoop : MonoBehaviour
{
    private float width; // 배경의 가로 길이

    private void Awake()
    {
        // 가로 길이를 측정하는 처리
    }

    private void Update()
    {
        // 현재 위치가 원점에서 왼쪽으로 width 이상 이동했을때 위치를 리셋
    }

    // 위치를 리셋하는 메서드
    private void Reposition()
    {

    }
}
```

3. 배경 스크롤

- **Awake() Method**에 **Collider Component**를 할당 받아서 **width**값을 알아오자.
  →사용자의 설정에 따라 **size**는 달라질 수 있지만 **Sprite**의 경우 초기 **Component**는 해당 **Sprite**의 **Texture**의 원본 **FixelSize** 정보를 가지고 있기 때문에 기본값으로 설정이 된다. 그것을 이용하는 것

```
private void Awake()
{
    //BoxCollider2D Component의 Size Field의 x 값을 가로 길이로 사용
    BoxCollider2D backgroundCollider = GetComponent<BoxCollider2D>();
    width = backgroundCollider.size.x;
}
```

Update() Method에 위치를 재설정하는 부분을 작성하자.

```
private void Update()
{
    // 현재 위치가 원점에서 왼쪽으로 width 이상 이동했을 때 위치를 재배치
    if (transform.position.x <= -width)
    {
        Reposition();
    }
}
```

Reposition() Method를 이용해서 **Position** 변경을 관리하자.

```
private void Reposition()
{
    //현재 위치에서 오른쪽으로 가로 길이 * 2만큼 이동
    Vector2 offset = new Vector2(width * 2f, 0);
    //Vector3 offset = new Vector3(width * 2f, 0);
    transform.position = (Vector2)transform.position + offset;
    //transform.position = transform.position + offset;
}
```

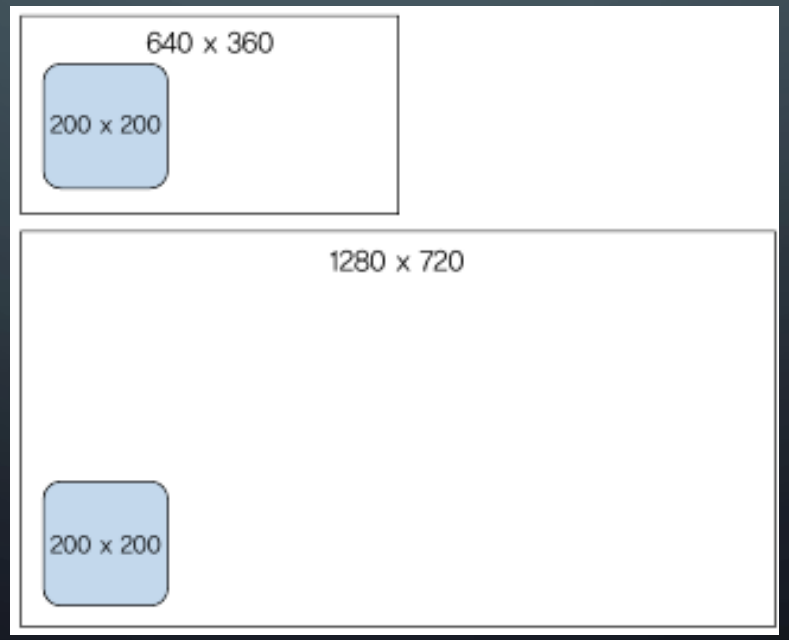
해상도에 맞는 UI

3. 게임 UI만들기

고정 픽셀 크기(Constant Pixel Size) VS 화면에 따른 스케일 변경(Scale With Screen Size)

고정 픽셀 크기

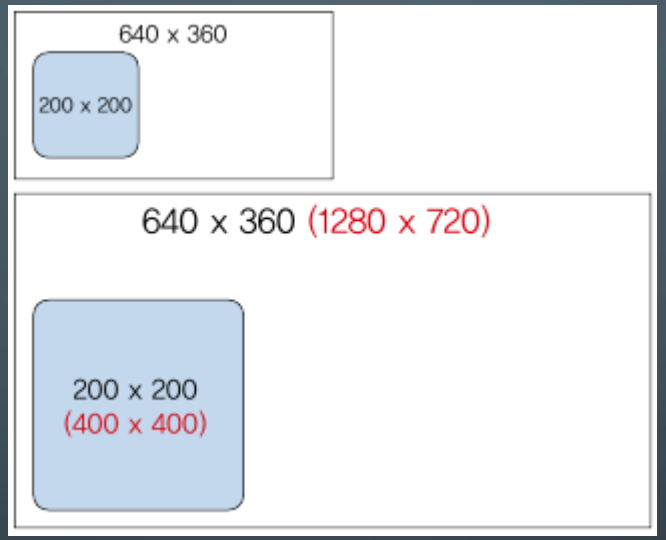
- **Canvas Scaler Component**의 **UI Scale Mode**를 **Constant Pixel Size**를 선택하면 적용된다.
- **Canvas**의 크기가 변경되어도 배치된 **UI** 요소의 크기를 변경하지 않는다.
  →화면의 크기가 달라지면 **UI** 요소의 크기나 **UI**요소 사이의 간격이 의도와 달게 크거나 작아지는 문제가 생긴다.
  →**640x360** 해상도를 기준으로 **UI**를 제작했는데 **1280x720**이 되면 상대적 크기와 위치가 고정이기 때문에 달라지게 된다.
- **UI**의 요소의 크기는 변화가 없지만 **Canvas**의 크기가 커져서 **UI**요소가 상대적으로 작게 보이는 문제가 생긴다.



3. 게임 UI만들기

화면 크기에 따른 스케일

- 실행화면이 기준 화면보다 크거나 작을 때는 자동으로 확대 / 축소하는 모드
  →**640x360** 해상도에서 **UI**를 배치했다고 가정하면 게임을 실행 중인 하면의 해상도가 **640x360**보다 크거나 작아도 무조건 **Canvas**의 크기를 **640x360**으로 취급한다**.**

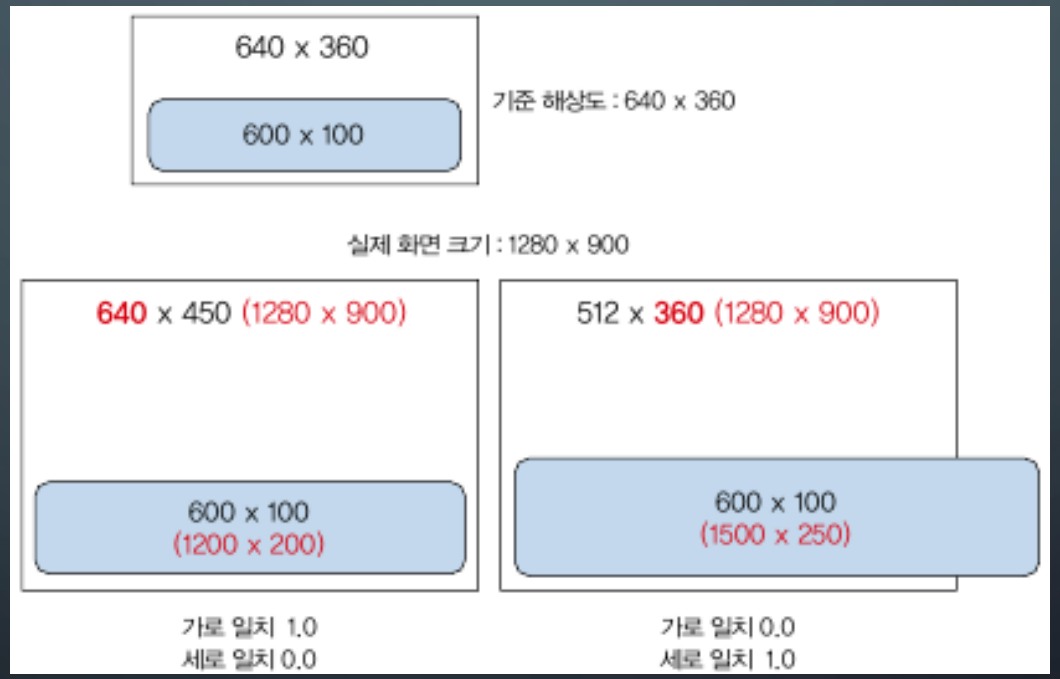


- 화면 해상도의 크기로 취급하는 것이 아니라 화면의 비율로 계산하기 때문에 해당 **UI**의 배치 간격과 화면에서 얼마큼 차지하고 있냐를 표현하기 때문에 화면의 크기가 두배가 커진다면 **UI**들도 비율을 맞춰야 하기 때문에 **2**배사이즈가 된다**.**

3. 게임 UI만들기

방향매치

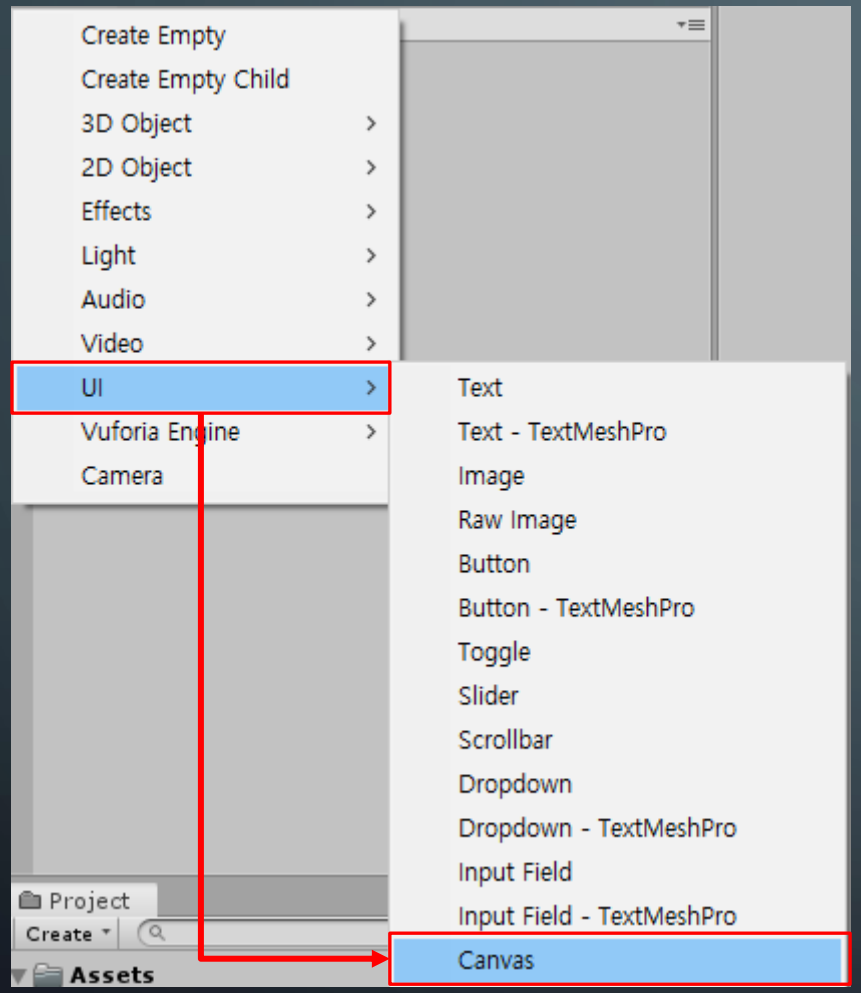
Scale With Screen Size Mode는 실제 화면과 기준 해상도 사이의 화면 비율이 다른 경우 **Canvas Scaler Component**의 **Match Field**값이 높은 방향의 길이를 유지하고 다른 방향의 길이를 조절한다.
→기준 해상도가 **640x360**일 때 가로로 긴 **600x100** 크기의 **UI**요소를 배치했다고 가정하면 **1280x900** 크기의 게임 화면에서 **Match Field**의 값에 따라 **Inspector View**에 표시되는 **Canvas**의 크기는 각각 다르다.



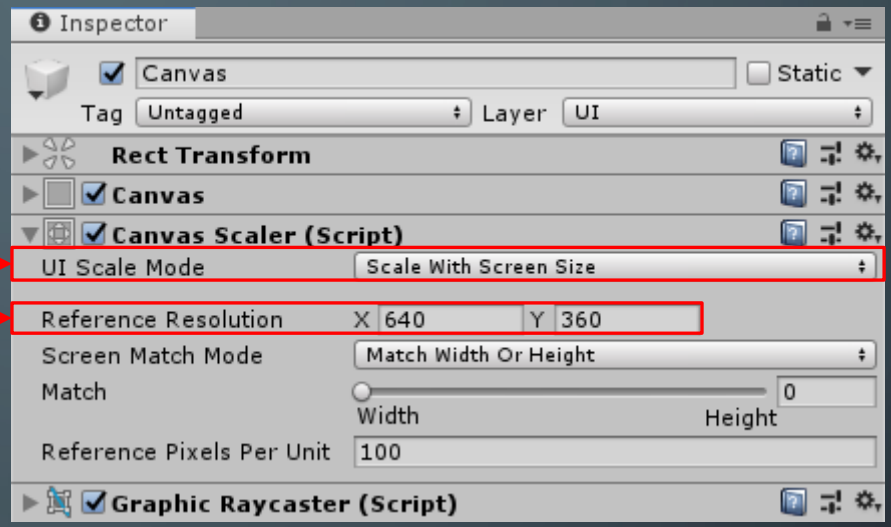
가로 일치가 **1.0**이면 **Canvas**의 가로 길이를 **640**으로 고정, 세로 길이를 변경
세로 일치가 **1.0**이면 **Canvas**의 세로 길이를 **360**으로 고정하고 가로 길이를 변경

3. 게임 UI만들기

Canvas 설치 및 설정 변경



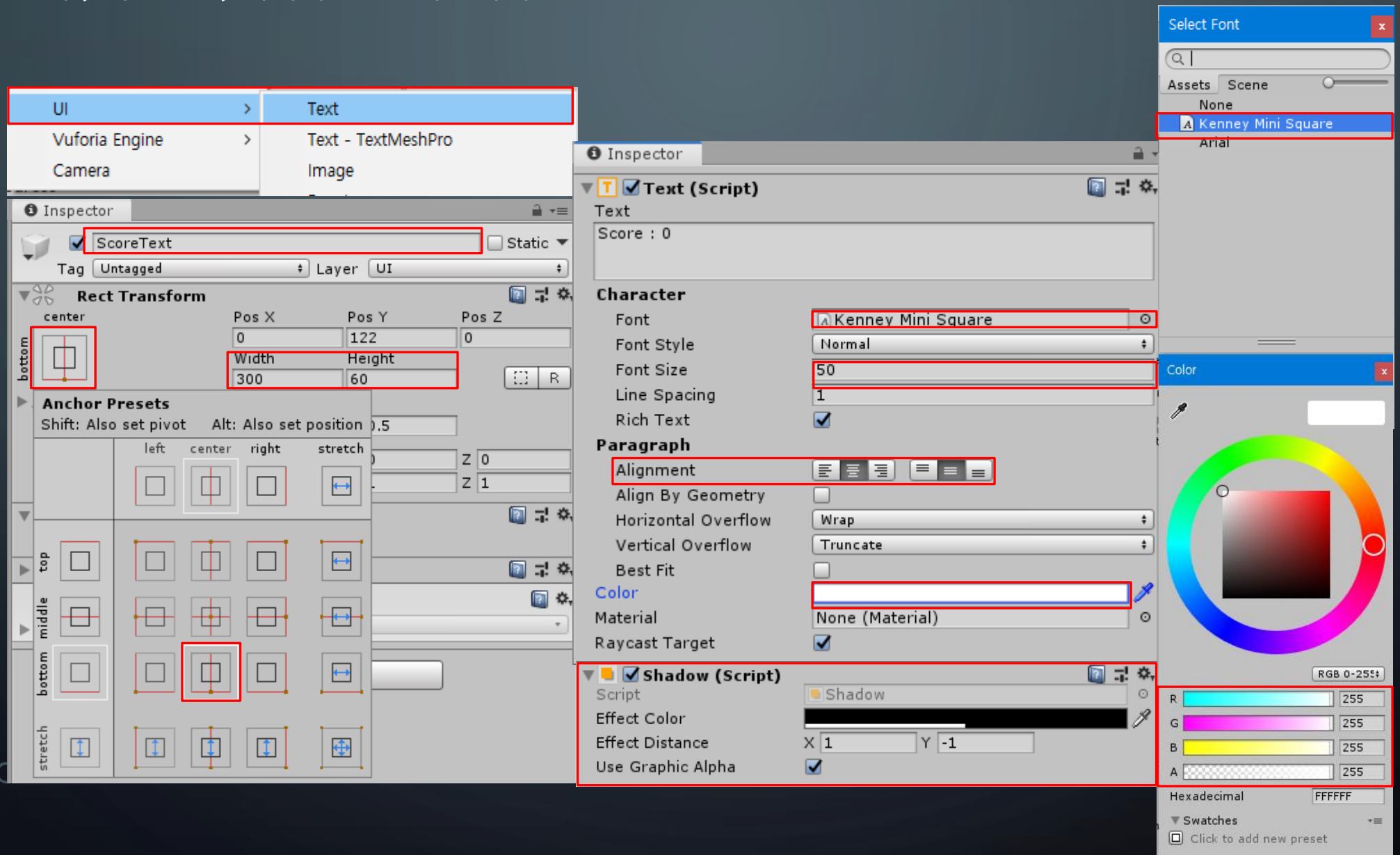
Canvas Scaler Component의 **UI Scale Mode**를 **Scale With Screen Size**로 변경



Reference Resolution을 **(640, 360)**으로 변경

3. 게임 UI만들기

점수, 게임종료, 재시작 **UI**를 작성하자.



4. GAMEMANAGER 만들기 - SINGLETON

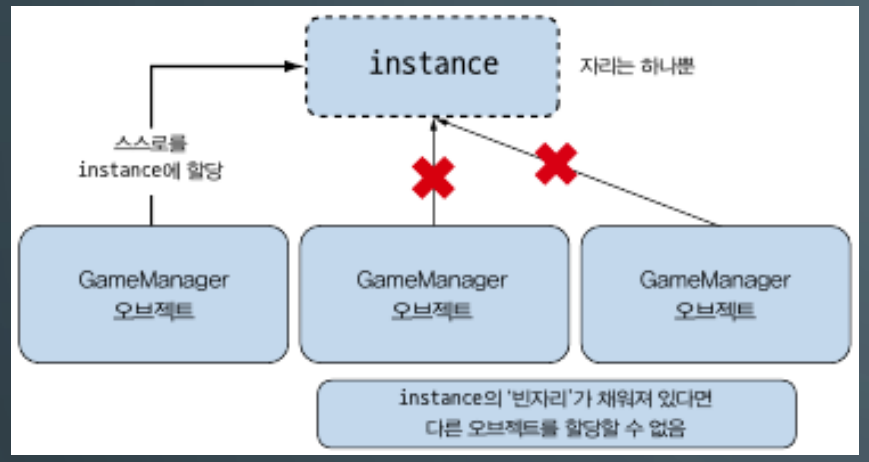
GameManager로 **Singleton**만들기

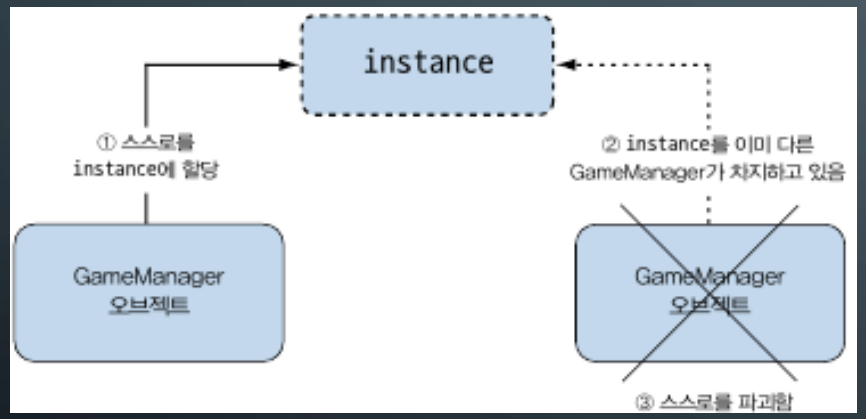
```
// 게임 오버 상태를 표현하고, 게임 점수와 UI를 관리하는 게임 매니저
// 씬에는 단 하나의 게임 매니저만 존재할 수 있다.
public class GameManager : MonoBehaviour
{
    public static GameManager instance; // 싱글톤을 할당할 전역 변수
// 게임 시작과 동시에 싱글톤을 구성
void Awake()
{
    // 싱글톤 변수 instance가 비어있는가?
    if (instance == null)
    {
        // instance가 비어있다면(null) 그곳에 자기 자신을 할당
        instance = this;
    }
    else
    {
        // instance에 이미 다른 GameManager 오브젝트가 할당되어 있는 경우

        // 씬에 두개 이상의 GameManager 오브젝트가 존재한다는 의미.
        // 싱글톤 오브젝트는 하나만 존재해야 하므로 자신의 게임 오브젝트를 파괴
        Debug.LogWarning("씬에 두개 이상의 게임 매니저가 존재합니다!");
        Destroy(gameObject);
    }
}
```

4. 게임 UI만들기

static으로 선언된 **instance**는 모든 오브젝트가 공유하는 단 하나의 변수가 된다**.**



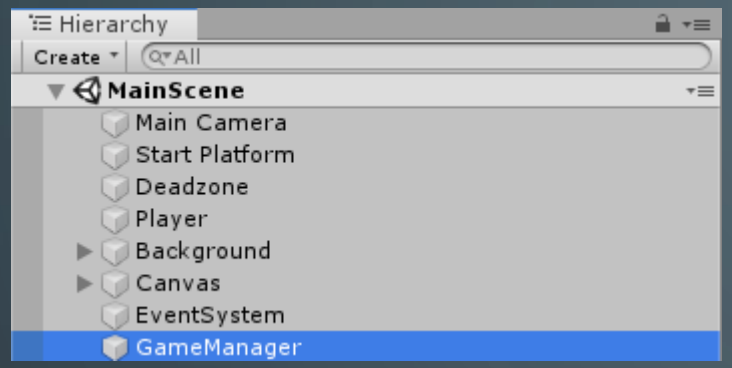


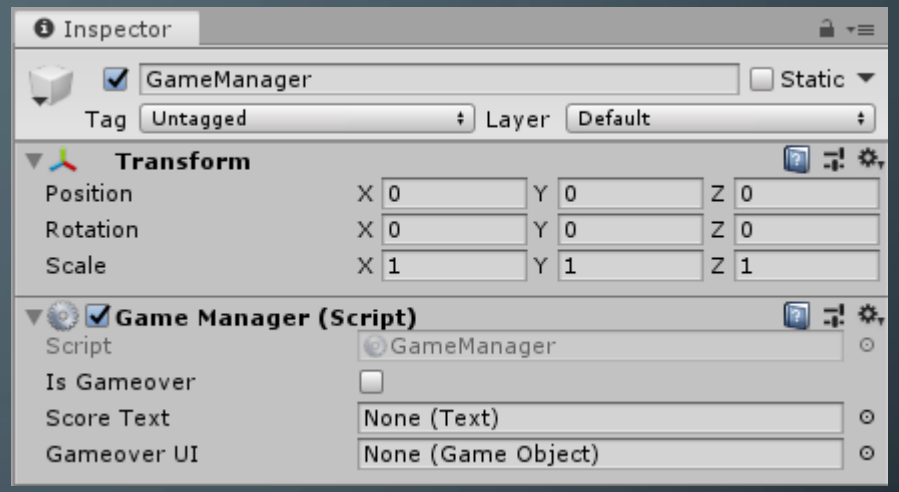
Awake() Method를 통해 **instanc**에 **GameManager** 자기 자신을 할당한다**.**
→이미 생성되었는데 또 **GameManager Component**가 추가된 **GameObject**가 생성되거나 활성화 되려고 한다면 이미 생성되었기 때문에 더 이상 생성할 수 없다고 하고 삭제하여 **GameObject**도 하나만 남기게 한다**.**

GameManager.instanc로 즉시 접근할 수 있다**.**

4. GAMEMANAGER 만들기 - SINGLETON

GameManager 오브젝트를 생성하고 **GameManager Component**를 생성하자.





Update() Method에서 **Gameover**상태가 **true**인지 확인해서 마우스 왼쪽 클릭 시, 재시작 코드를 넣는다. → 현재 활성화 중인 **Scene**을 체크해서 **Scene**이름을 알아오고 다시 로드한다.

```
void Update()
{
    //게임오버 상태에서 마우스 왼쪽 버튼을 클릭하면 현재 Scene Restart
    if(isGameover && Input.GetMouseButtonDown(0))
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
}
```

4. GAMEMANAGER 만들기 - SINGLETON

AddScore() Method를 완성하고 적용해보자.

```
public void AddScore(int newScore)
{
    //게임 오버가 아니라면
    if (!isGameover)
    {
        //점수를 증가
        score += newScore;
        scoreText.text = "Score : " + score;
    }
}
```

OnPlayerDead()를 작성하고 **Player**의 **Die() Method**에서 동작하게 하자.

```
public void OnPlayerDead()
{
    isGameover = true;
    gameoverUI.SetActive(true);
}
```

```
private void Die()
{
    //애니메이터의 Die 트리거 파라미터를 셋팅한다.
    animator.SetTrigger("Die");

    //오디오 소스에 할당된 오디오 클립을 deathClip으로 변경
    playerAudio.clip = deathClip;
    //사망 효과음 재생
    playerAudio.Play();

    //속도를 제로(0, 0)로 변경
    playerRigidbody.velocity = Vector2.zero;
    //사망 상태를 true로 변경
    isDead = true;

    GameManager.instance.OnPlayerDead();
}
```

4. GAMEMANAGER 만들기 - SINGLETON

ScrollingObject의 **Update() Method**를 종료 했을 때로 수정해보자.

```
private void Update()
{
    // 게임오버가 아니라면
    if (!GameManager.instance.isGameover)
    {
        // 초당 speed의 속도로 왼쪽으로 평행이동
        transform.Translate(Vector3.left * speed * Time.deltaTime);
    }
}
```

GameManager의 **Field**를 할당해주고 실행 해보자