UNITY
-CAHPTER4-1-

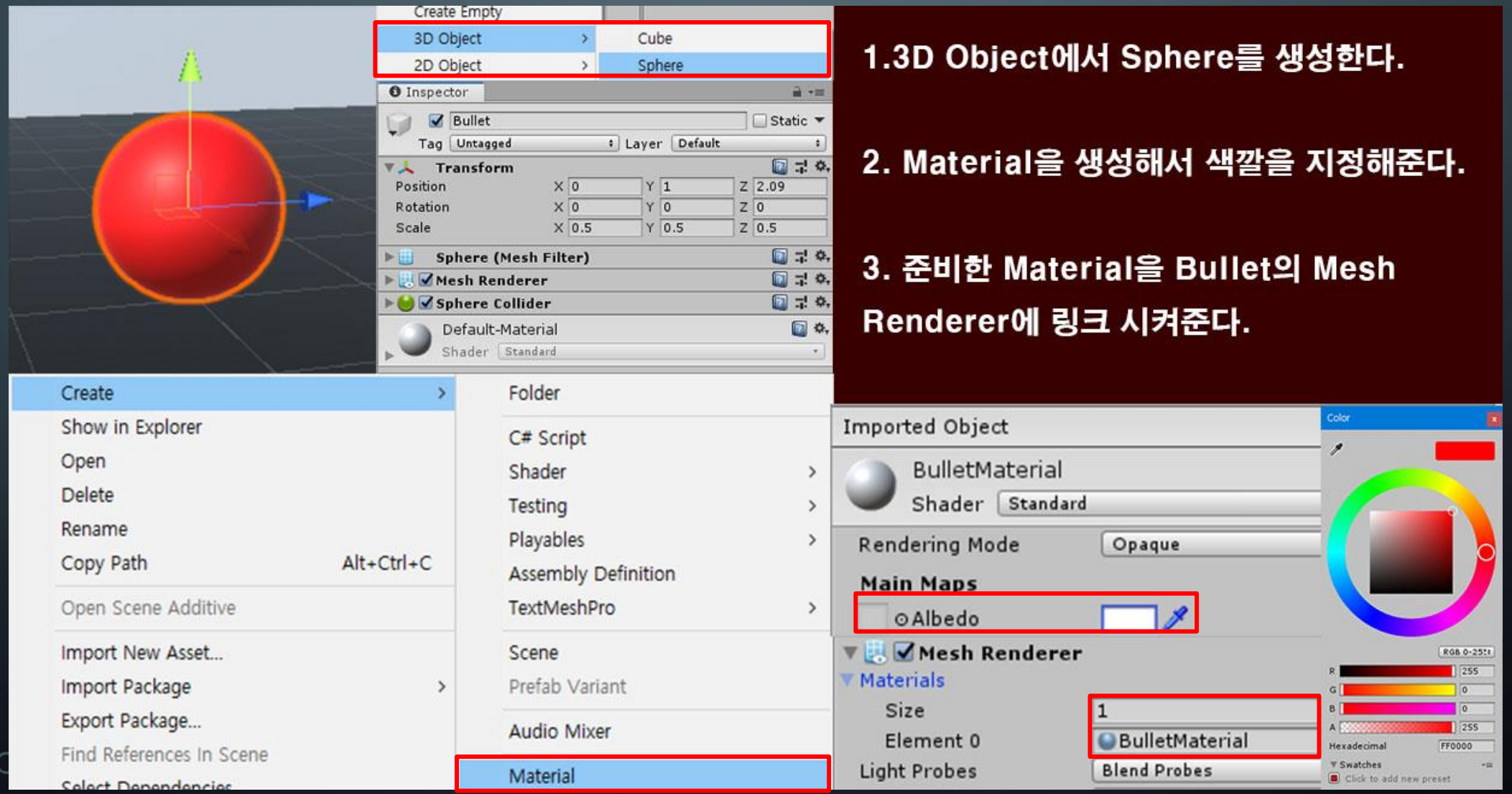
SOUL SEEK

목차

탄알제작과 충돌처리

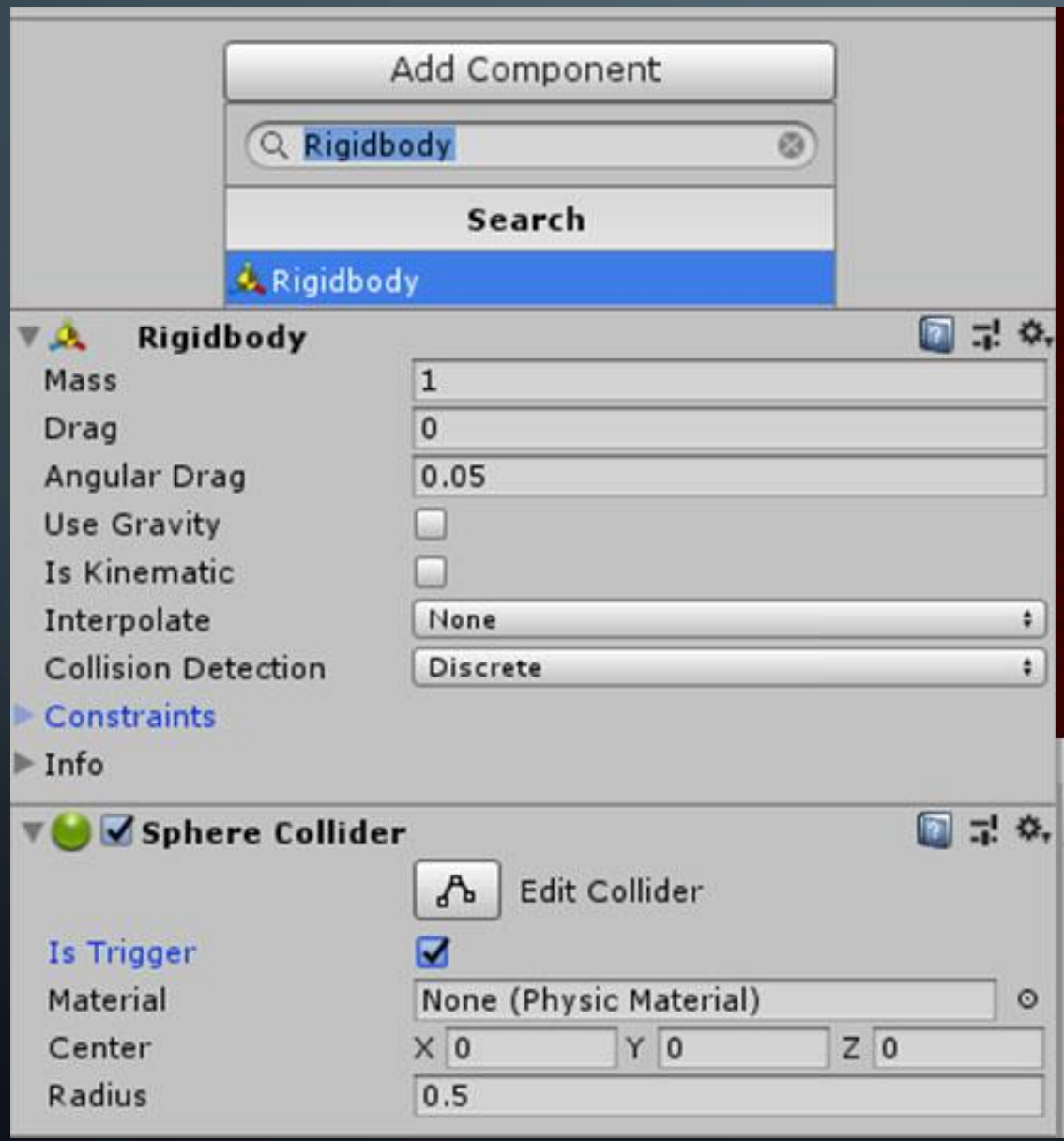
1. 탄약제작과 충돌처리

Bullet Object 생성



1. 탄약제작과 충돌처리

Bullet Prefab 생성

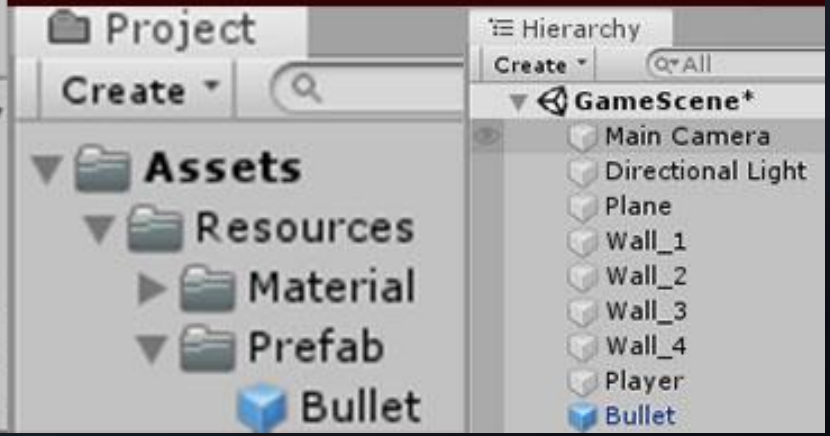


1.Rigidbody를 추가한 다음 Rigidbody 속성에서 Use Gravity 체크를 해제 해준다

->중력값을 받게 되면 아래로 계속 떨어지기 때문이다.

2.Collider에서 Is Trigger 속성을 체크해서 이벤트 체크를 진행한다.

3.Component가 추가 된 Bullet을 Prefab으로 생성한다.



1. 탄약제작과 충돌처리

Bullet Script 생성

- **Script**를 생성하고 **speed**와 **Rigidbody**를 멤버 변수로 준비한다.

```
public float speed = 8f;
private Rigidbody bulletRigidbody;
```

- **Start() Method**에 **bulletRigidbody**에 자신의 **Rigidbody Component**를 할당하고 **Velocity**를 설정해 준다.

```
void Start()
{
    // 게임 오브젝트에서 Rigidbody Component를 찾아 bulletRigidbody에 할당
    bulletRigidbody = GetComponent<Rigidbody>();
    // Rigidbody의 속도 = 앞쪽 방향 * 이동 속력
    bulletRigidbody.velocity = transform.forward * speed;
}
```

- **Start()**에 **Destroy()** 메소드를 이용해 **gameObject**를 파괴 예약을 한다.

```
//3초 뒤에 자신의 gameObject 파괴
Destroy(gameObject, 3f);
```

- **Destroy(gameObject);** 라고 하면 즉시 파괴가 일어난다.

1. 탄약제작과 충돌처리

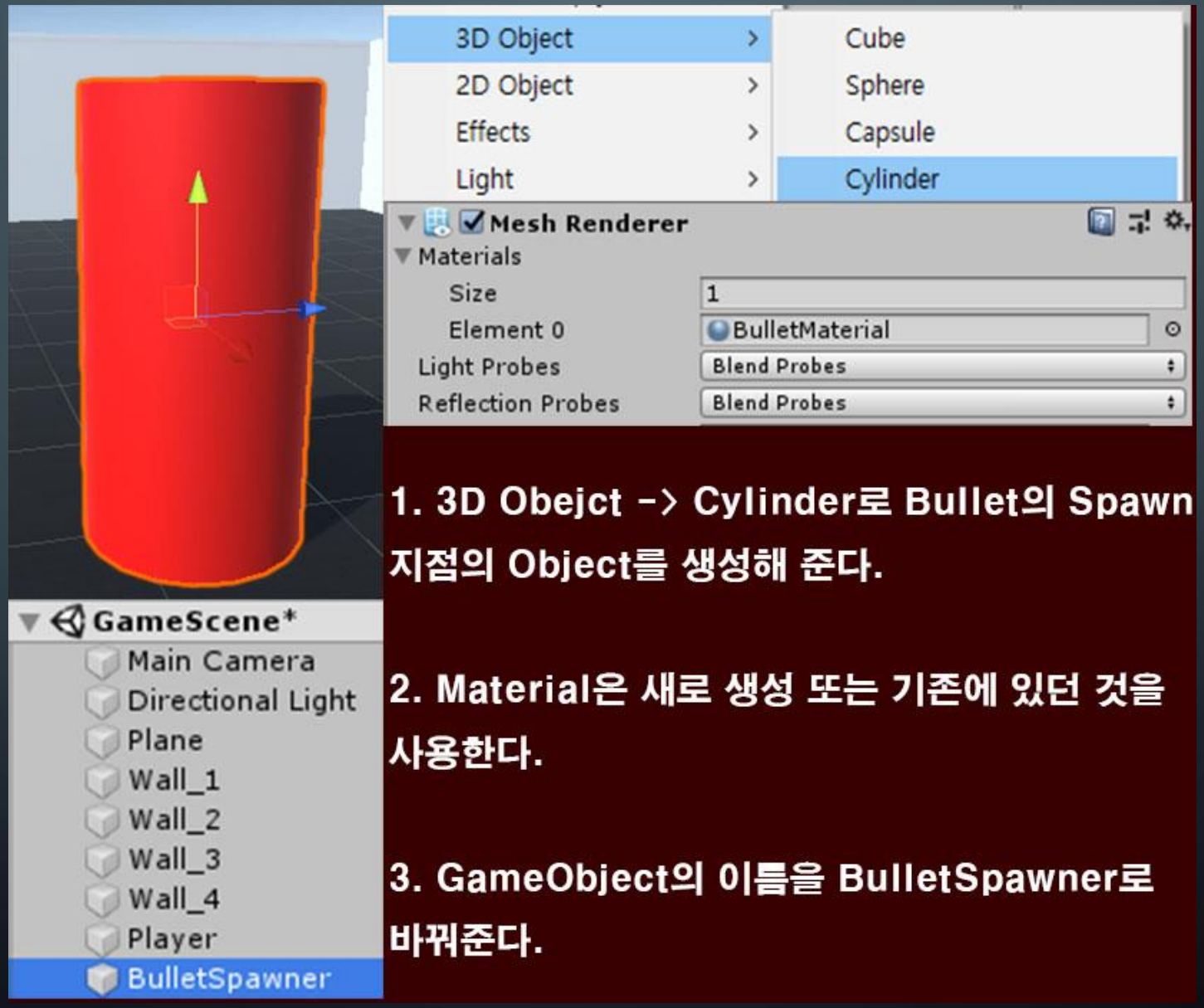
- **Is Trigger**을 체크 했기 때문에 충돌 이벤트만 체크 할 수 있기 때문에 **OnTriggerXXX() Method**로 체크 할 수 있다.

```
//Trigger 충돌 시 자동으로 실행되는 Method
private void OnTriggerEnter(Collider other)
{
    //충돌한 상대방 GameObject가 Player 태그를 가진 경우
    if(other.tag == "Player")
    {
        //상대방 GameObject에서 PlayerController Component 가져오기
        PlayerController playerController = other.GetComponent<PlayerController>();

        //상대방으로부터 PlayerController Component를 가져오는 데 성공했다면
        if(playerController != null)
        {
            //상대방 PlayerController 컴포넌트의 Die() Method 실행
            playerController.Die();
        }
    }
}
```

1. 탄약제작과 충돌처리

Bullet 생성기 제작(탄알 생성 지점)



1. 탄약제작과 충돌처리

BulletSpawner Script를 생성하고 편집하자.

- **BulletSpawner**는 **Bullet**의 생성과 생성타임을 조절할 것이기 때문에 **Bullet Prefab**을 담을 객체와 딜레이 타임을 조절하는 객체를 들고 있고 **Player**를 향해 발사 해야 하기 때문에 **Player**의 위치를 받기 위한 **Transform** 객체도 들고 있다.

```
public class BulletSpawner : MonoBehaviour
{
    public GameObject bulletPrefab; //생성할 탄알의 원본 Prefab
    public float spawnRateMin = 0.5f; //최소 생성주기
    public float spawnRateMax = 3f; //최대 생성주기

    private Transform target; //발사대상
    private float spawnRate; //생성주기
    private float timeAfterSpawn; //최근 생성 시점에서 지난 시간
```

- **Start() Method**에서 생성 체크 시간을 초기화 하고 딜레이 간격을 위한 랜덤 값을 설정하고 최초 타겟 지점을 설정한다.

```
void Start()
{
    //최근 생성 이후의 누적 시간을 0으로 초기화
    timeAfterSpawn = 0f;
    //탄알 생성 간격을 spawnRateMin과 spawnRateMax 사이에서 랜덤 지정
    spawnRate = Random.Range(spawnRateMin, spawnRateMax);
    //PlayerController Component를 가진 GameObject를 찾아 조준 대상으로 설정
    target = FindObjectOfType<PlayerController>().transform;
}
```

1. 탄약제작과 충돌처리

FindObjectOfType() Method

- 지정한 **Type(Component)**이 존재하는 **GameObject**를 찾아서 **gameObject**를 반환한다**.**
- **Scene**에 존재하는 모든 **Object**를 검색하여 원하는 **Type**의 **Object**를 찾아 낸다**.**
- 처리 비용이 크기 때문에 **Start()**에서 한번만 찾아주는 것이 좋다**.** ➔ **Update() Method에서 사용 X**
- **FindObjectsOfType()**이라는 **Method**로 원하는 타입을 가진 모든 **GameObject**를 배열형태로 제공 한다**.**

Update() Method에서 **Bullet**을 일정 간격으로 생성

```
void Update()
{
    //timeAfterSpawn 갱신
    timeAfterSpawn += Time.deltaTime;

    //최근 생성 시점에서 누적된 시간이 생성 주기보다 크거나 같다면
    if(timeAfterSpawn >= spawnRate)
    {
        //누적된 시간을 리셋
        timeAfterSpawn = 0f;
        //bulletPrefab의 복제본을 transform.position위치와 transform.rotation 회전으로 생성
        GameObject bullet
            = Instantiate(bulletPrefab, transform.position, transform.rotation);
        //생성된 bullet GameObject의 정면 방향이 target을 향하도록 회전
        bullet.transform.LookAt(target);
        //다음번 생성 간격을 spawnRateMin, spawnRateMax 사이에서 랜덤 저장
        spawnRate = Random.Range(spawnRateMin, spawnRateMax);
    }
}
```

1. 탄약제작과 충돌처리

Time.deltaTime

FPS는 컴퓨터의 성능에 따라 다르기 때문에 각 **Frame** 간격동안 흐른 시간이 일정하지 않기 때문에 **Update()** 간 반복주기가 일정하지 않다.
매 **Frame** 반복된 시간동안 시간 간격을 알아 내기 위해 사용한다.
Update() Method에서 어떤 객체에 **Time.deltaTime** 값을 계속 누적하면 특정 지점으로 부터 시간이 얼마나 흘렀는지 표현할 수 있다.

```
timeAfterSpawn += Time.deltaTime;
```

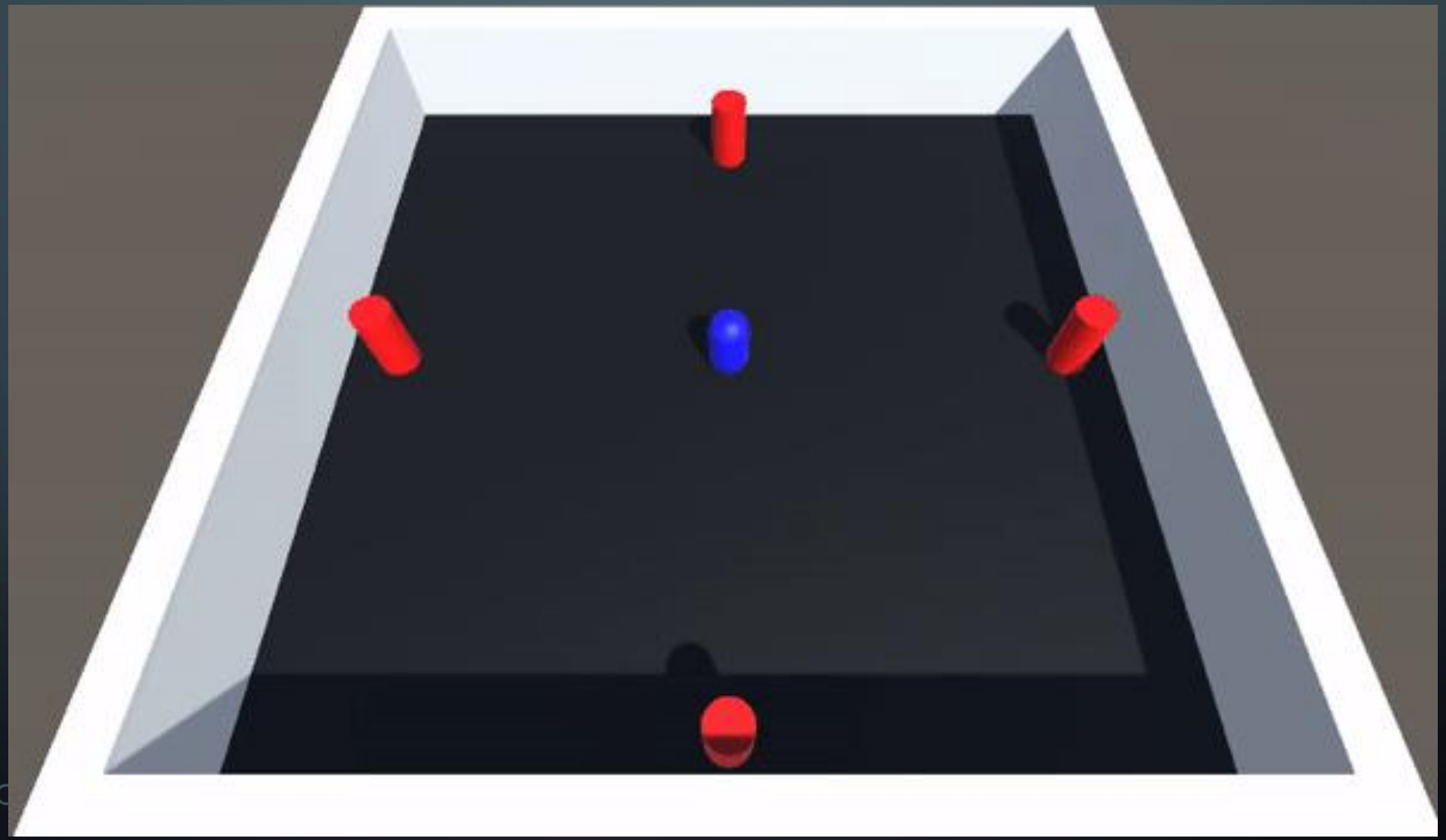
Update() Method가 진행 될 때마다 **timeAfterSpawn**에 시간이 누적된다.

```
if(timeAfterSpawn >= spawnRate)
```

timeAfterSpawn을 **Random Method**에 의해 만들어진 **spawnRate**와 비교해서 값이 크거나 같으면 **timeAfterSpawn**을 초기화 하고 **Bullet**을 생성한 뒤에 다시 **spawnRate**를 설정한다.

이렇게 완성된 **Script**를 **BulletSpwaner**에 추기하고 **Prefab**으로 만든 뒤 **Scene**에 적절히 배치하고 실행 시켜보자.

1. 탄약제작과 충돌처리

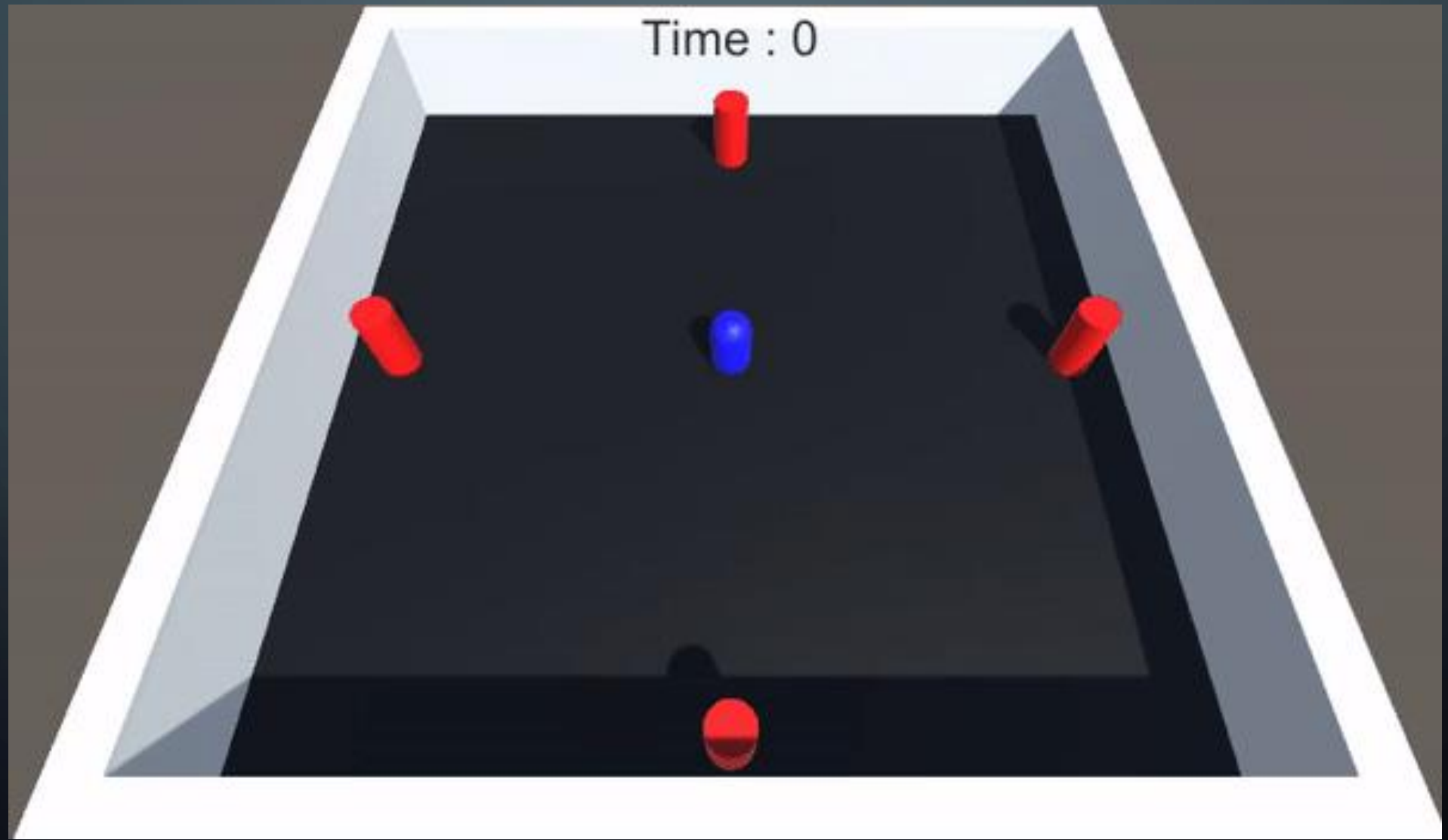


UI(USER INTERFACE)

2. UI(USER INTERFACE)

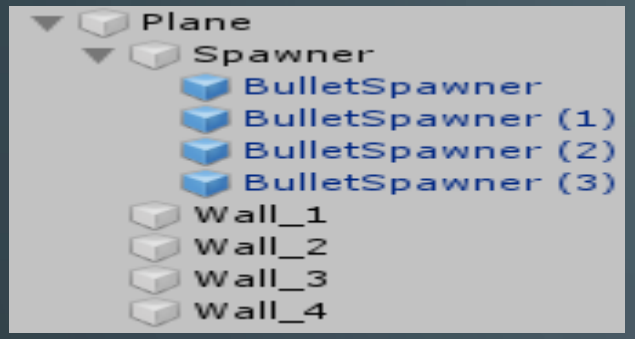
Level 요소 추가 – 게임 난이도 요소

Ex) Level Design 중 환경요소들을 **Ratate** 시켜서 난이도를 증가시킨다**.**



2. UI와 GAMEMANAGER

단계 **1.** 회전 시키고자 하는 **GameObject**를 선택한다**.**



단계**2.** 회전을 하게 하는 최상위 **GameObject**에게 추가할 **Compoenent**를 제작하자**.**

단계**3. transform.Rotate() Method**를 이용해서 매 프레임마다 회전하게 **Component**를 편집하자**.**

```
float rotationSpeed = 8f;

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

단계**4.** 준비된 **Component**를 추가하자**.**

2. UI(USER INTERFACE)

InGameUI제작

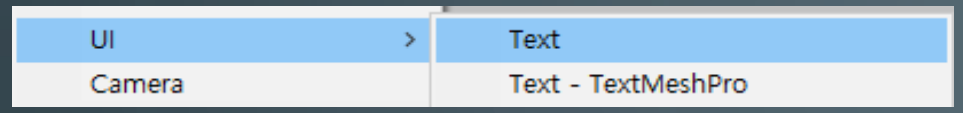
- 게임 월드와 공간개념이 다르므로 **UI**는 별도의 공간으로 처리한다**.**
- **Scene**상에서 보여지는 것이 그대로 **Game View**에 나타나는 것이 아니므로 안심해도 된다**.**
- **Unity**에서는 **NGUI(Next gen UI)**라는 외부 **Plugin**에서 착안한 **UGUI**를 자체적으로 지원 하고 있다**.**

UGUI

- **UNITY**에서 제공하는 **UI** 오브젝트 관리 시스템**.**
- **NGUI**의 기능을 많이 가지고 있으며**, NGUI**와 똑같은 문제인 **Particle Randerring Sort** 문제가 있다**.**
- **Hierarchy**상의 계층 순서대로 **Depth**를 처리하기 때문에 **NGUI**보다 신경을 덜 쓸 수 있다**.**
- **Canvas, Image, Text**를 알아보자**.**
- **Script**에서 **UnityEngine.UI;**를 선언하고 **Code**에서 컨트롤 할 수 있다

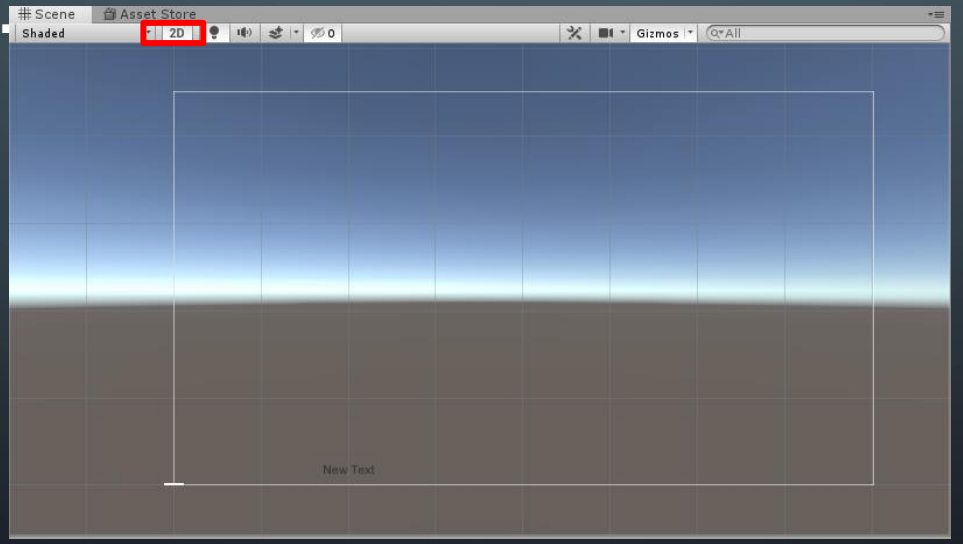
2. UI(USER INTERFACE)

생존 시간 체크용 **UI**를 추가해 보자.



- **UI→Text**로 **Text**를 추가 할 수 있는 **Text UI Object**가 생성된다.

- **UGUI**에서는 모든 **UI Object**들은 **UI**가 그려지고 배치되어질 공간인 **Canvas**가 반듯이 하나 이상은 필요하기 때문에 **Canvas**가 없다면 **Canvas**가 자동으로 추가되고 **Canvas**를 **Root**로 추가한 **UI Object**가 추가된다.

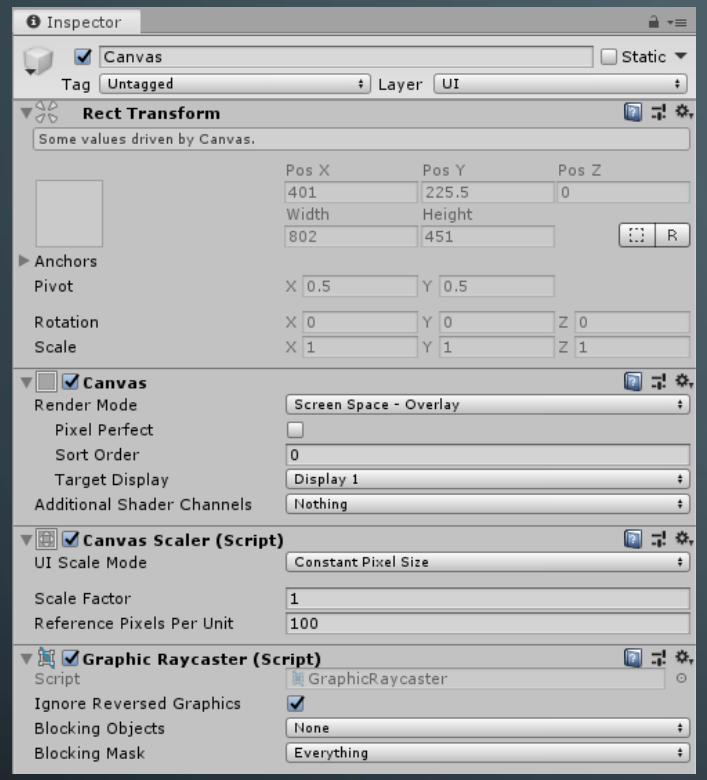


- 우리가 제작한 **3D Object**와 별도로 **Canvas**라는 공간안에서 모든 작업이 진행된다.
- **Canvas**는 하위 **UI Object**의 **Anchor** 를 관리하고 속성 설정에 따라 **Main Camera**에 어떻 형태로 나타낼 것 인지가 달라 진다.

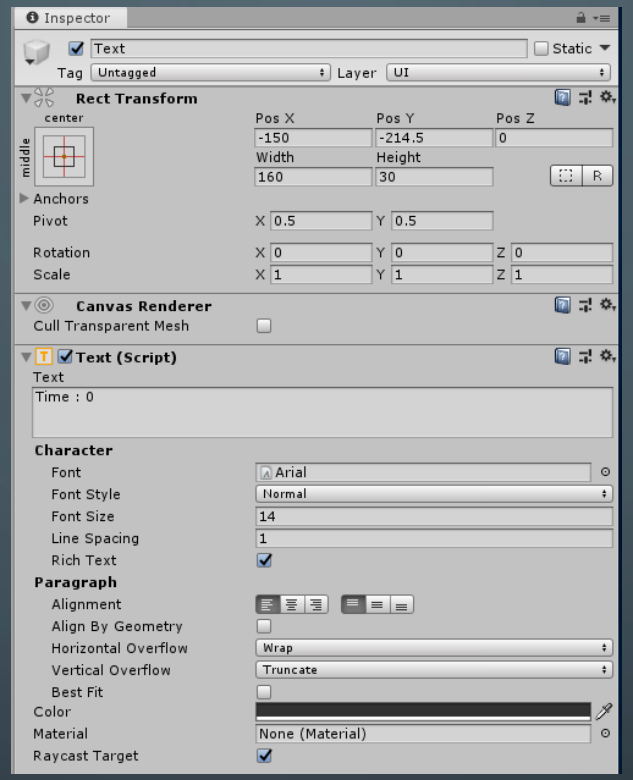
- **Scene**으로 돌아가서 **2D** 버튼을 활성화해서 **2D Mode**로 변경해보자.
  ➔ **UI** 작업에서는 **2D Mode**로 **Y**축 회전을 고정해주는 것이 작업이 편하다.

2. UI(USER INTERFACE)

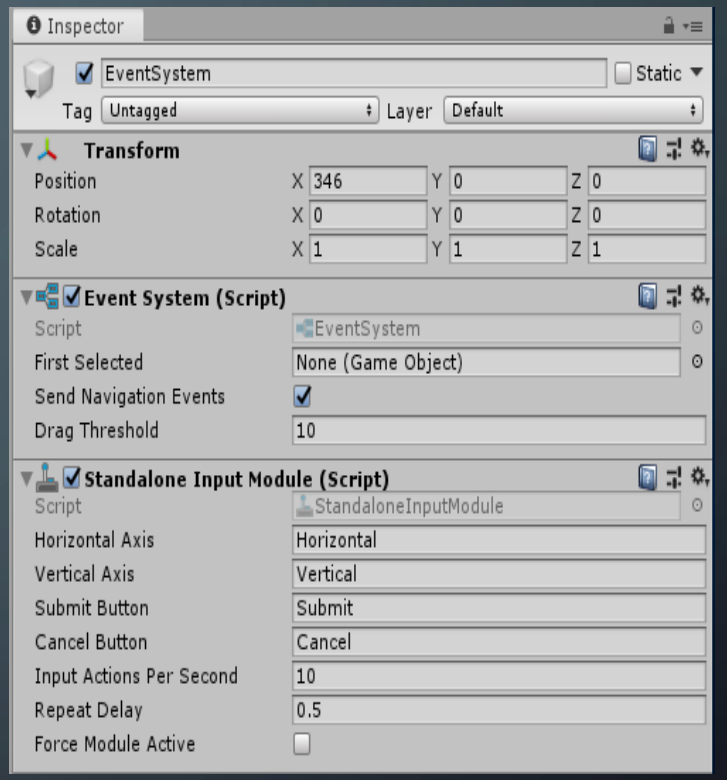
Scene에 추가된 각각의 **Object**들을 보면 여러가지 **Component**들이 추가 되어 있다.



Canvas의 속성들



Text의 속성들

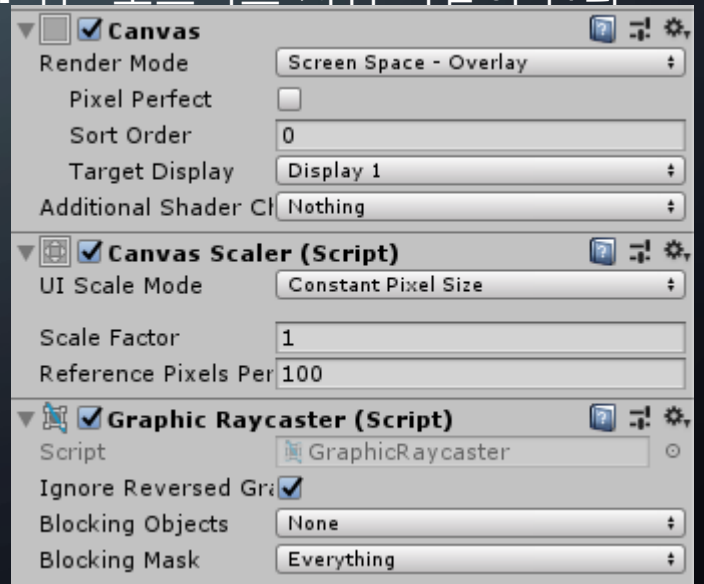


EventSystem의 속성들

2. UI(USER INTERFACE)

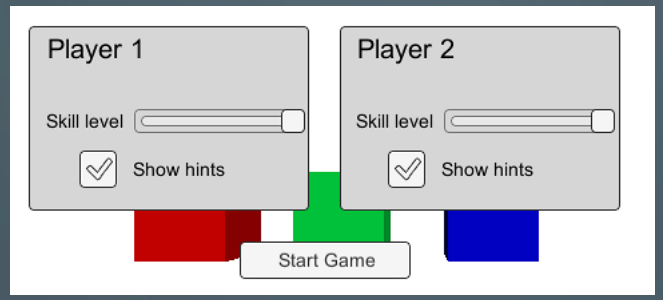
- **Canvas**
  - 모든 **UI** 요소를 배치하기 위한 영역**. Canvas** 요소와 함께 사용하는 게임 오브젝트로**,** 모든 **UI** 요소는 **Canvas**의 자식 요소여야 한다**.**
  - **UI**요소들을 그룹화하는 요소**, UI**요소들은 **Canvas**위에 존재해야 한다**.**
  - 한 씬에서 여러개의 **Canvas**가 존재할 수 있다**.**
  - **UI**요소들은 **Canvas**의 자식으로 존재한다**.**

- **Render Mode :**
  - **Screen Space Overlay – Renderring** 되는 것들의 위에 **UI**가 온다**,** 강제로 화면 전체를 **Rect**로 채우기 때문에 **RectTransform**을 컨트롤 할 수 없다**.**
  - **Screen Space Camera – Scene**의 특정 카메라에 투영 시킬 때 사용한다**,** 나머지는 **Overlay**랑 같다**.**
  - **World Space : Scene** 볼륨의 **Eliment**를 **Renderring**한다.다른 오브젝트 처럼 똑같이 관리 한다**.**
- **Pixel Perfect :** 가장 가까운 픽셀로 조정**.**
  Render Mode에서 **Screen Space**를 선택했을 때 나타난다**.**

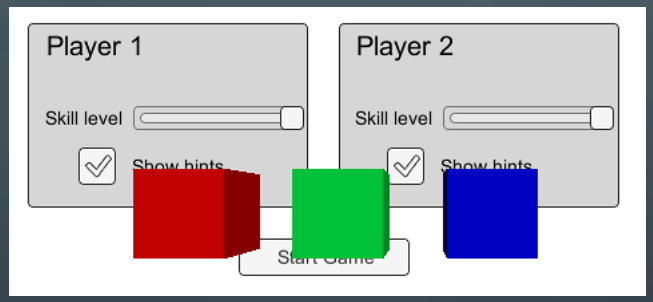


2. UI(USER INTERFACE)

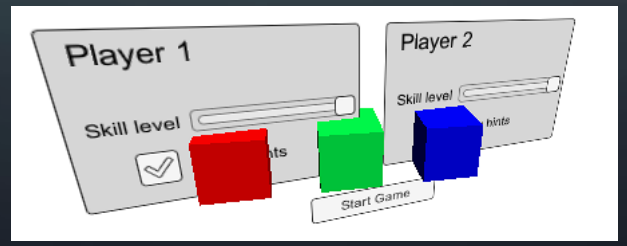
- **Screen Space – Overlay**



- **Screen Space – Camera**



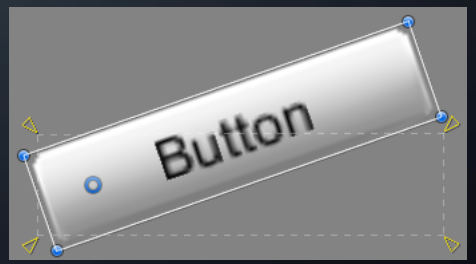
- **World Space**



2. UI(USER INTERFACE)

- **RectTransform**
  - **Transform**과 다른 직사각형 형태의 **UI**를 위한 **Transform Component**
  - **Pivot Point, Width, Height + Position, Rotate, Scale**
  - **Canvas**를 생성하게 되면 **RectTransform**은 자동으로 추가된다.
  - **Anchor**개념이 포함되어 있다.
  - **UI**요소들은 생성할 때마다 **RectTransform**을 가지고 있다.

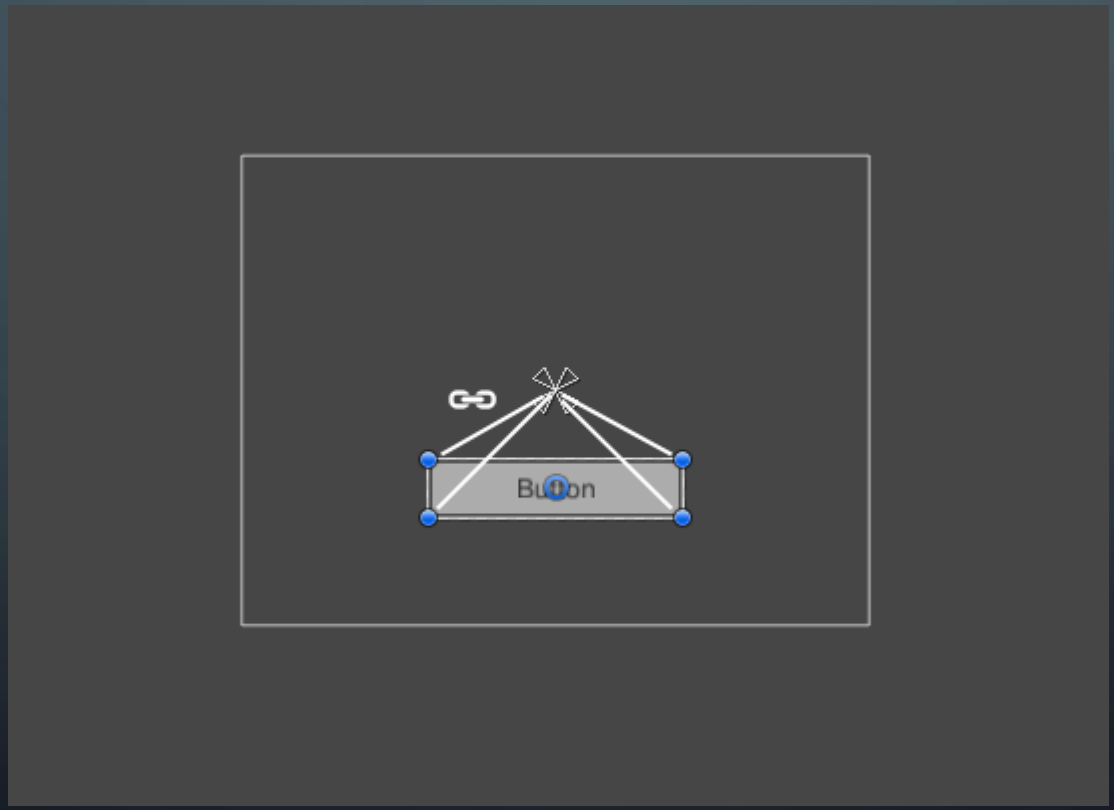
- **리사이징 VS 스케일링**
  - **Rect Tool**이 오브젝트의 크기 변경에 사용되는 경우, **2D** 시스템의 **Sprite**와 **3D** 오브젝트를 위해 일반적으로 오브젝트의 로컬 _**scale**_을 변경한다. 그러나 **Rect Transform** 컴포넌트가 연결된 오브젝트의 경우, 로컬 **scale**은 변경하지 않은 채 **width**와 **height**를 변경합니다. 이 리사이징은 글꼴 크기, 슬라이스 된 이미지의 경계선 등에 영향을 주지 않는다.



- **피벗(Pivot)**
  - 회전**(rotation)**, 크기**(size)**, 스케일**(scale)**의 수정은 피벗 주위에서 발생하므로 피벗의 위치는 회전, 리사이징 스케일링의 결과에 영향을 준다. 툴바의 **Pivot** 버튼이 **Pivot** 모드로 설정되어 있으면 **Rect Transform** 피벗은 **Scene View**에서 이동시킬 수 있다.

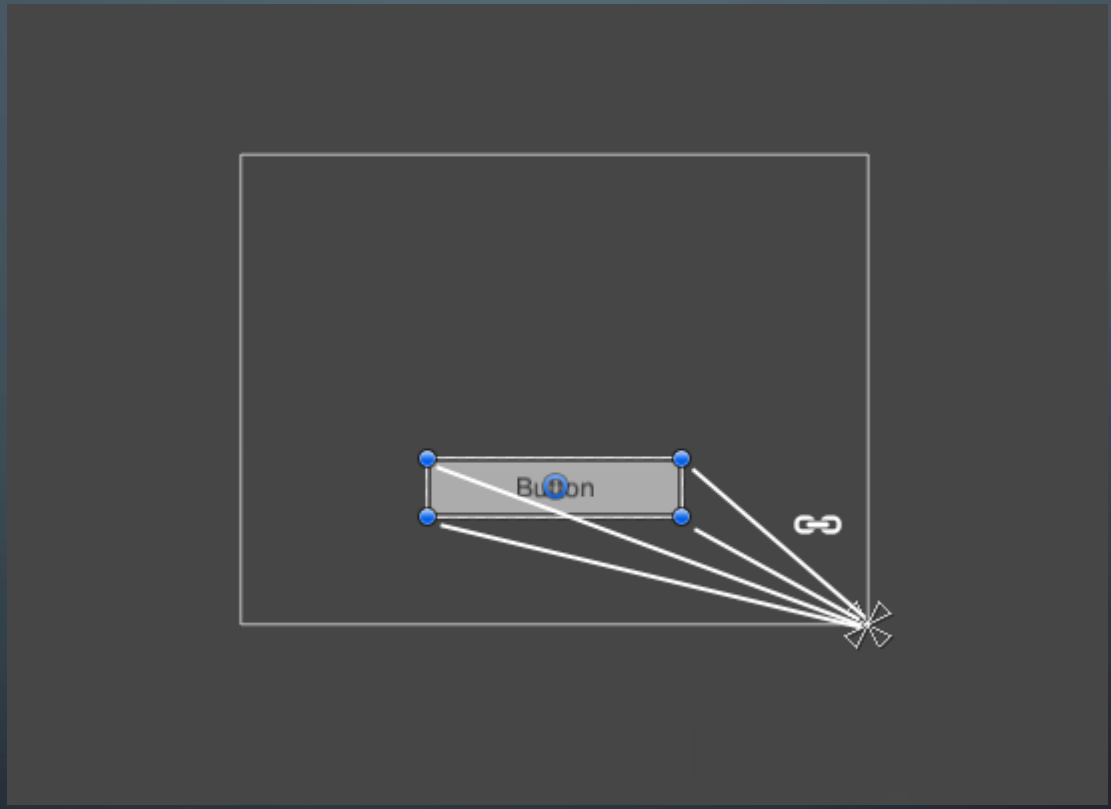
2. UI(USER INTERFACE)

- **Anchor**
  - **Rect Transform**은 **anchors**라는 레이아웃의 개념이 있다**.** 앵커는 **4**개의 작은 삼각형 핸들로 **Scene View**에 표시되고 앵커의 정보는 인스펙터에 표시된다**.**
  - **Rect Transform**의 부모도 **Rect Transform**이면 자식의 **Rect Transform**은 다양한 방법으로 부모의 **Rect Transform**에 고정할 수 있다**.** 예를 들어**,** 자식은 부모의 중심 또는 모서리 중 하나에 고정할 수 있다**.**



부모의 중심에 고정된 UI 요소. 요소는 중심에 대해 고정 오프셋을 유지하고 있다.

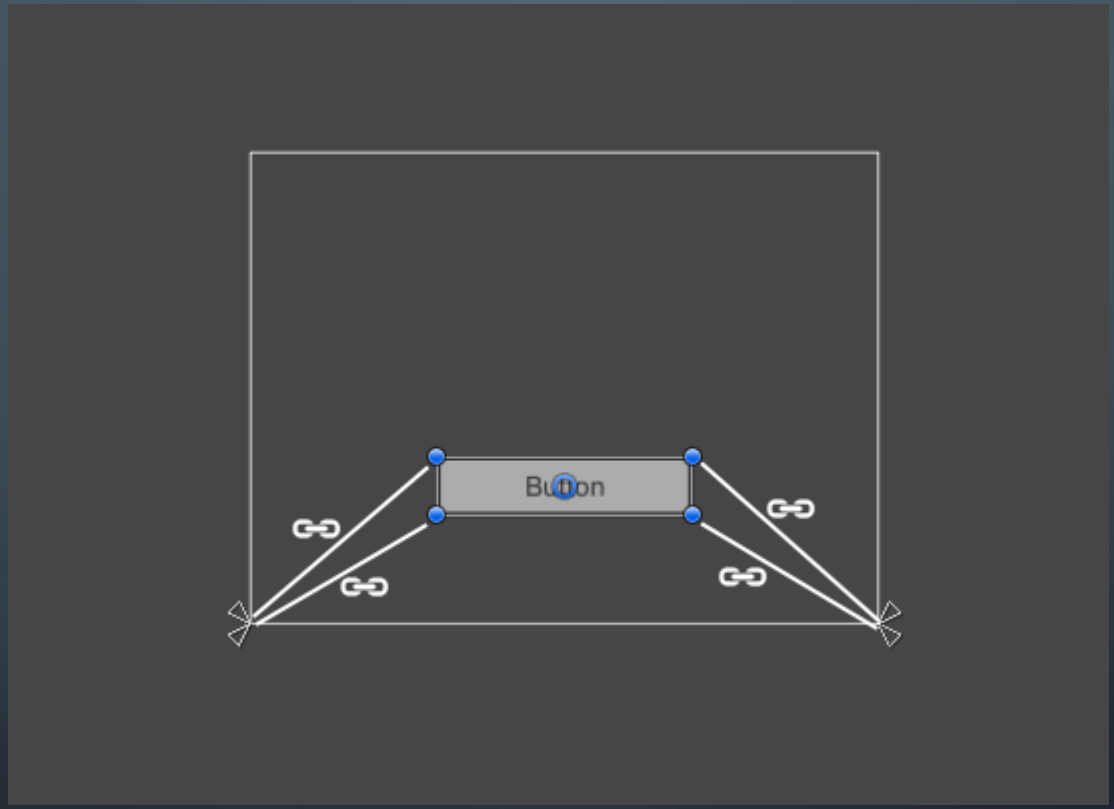
2. UI(USER INTERFACE)



부모의 오른쪽 하단에 고정된 UI 요소. 요소는 오른쪽 하단에 고정 오프셋을 유지하고 있다.

2. UI(USER INTERFACE)

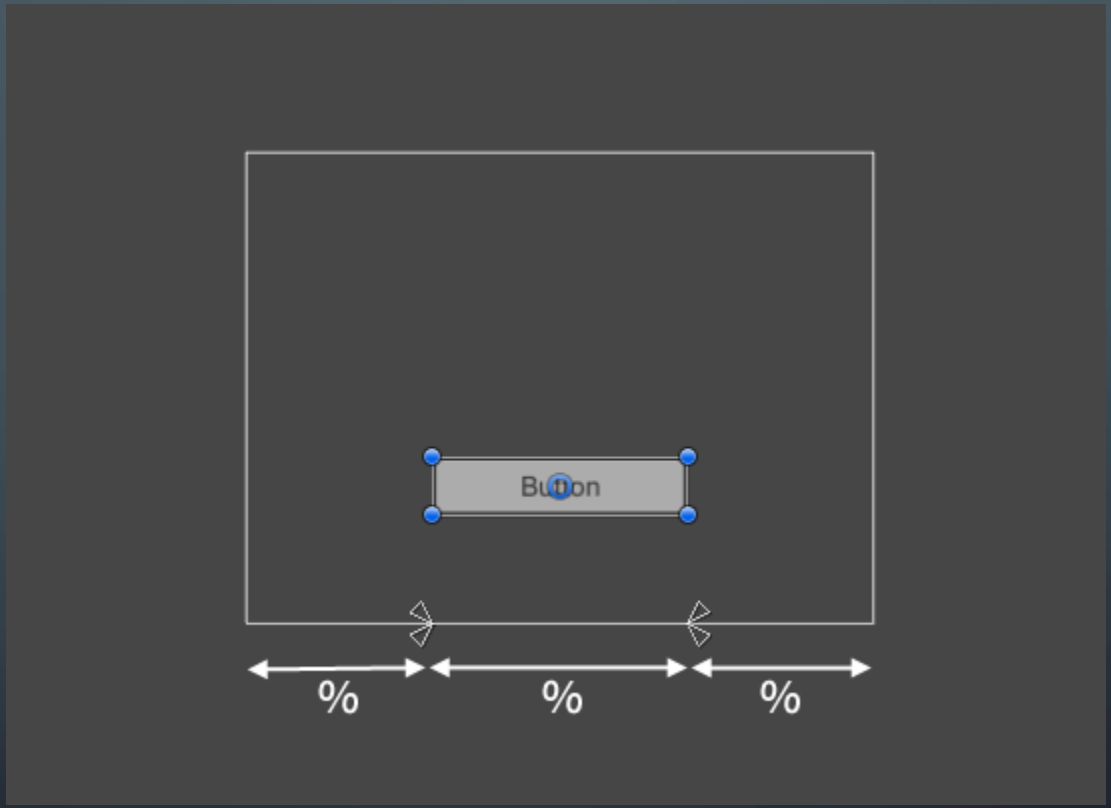
- 고정**(anchoring)**은 부모의 **width** 또는 **height**와 함께 자식을 늘리는 것을 허용한다**.** 사각형의 각 모서리는 해당 앵커에 고정 오프셋을 가지고 있다**.** 즉**,** 구형의 왼쪽 상단 모서리는 왼쪽 상단 앵커에 고정 오프셋을 갖는 것이다**.** 이와 같이**,** 사각형의 다른 모서리를 부모 사각형의 다른 점에 고정할 수 있다**.**



왼쪽 모서리를 부모 사각형의 왼쪽 아래 오른쪽 모서리가 오른쪽 하단 모서리에 고정된 UI 요소. 요소의 모서리는 각각의 앵커에 고정된 오프셋을 유지합니다.

2. UI(USER INTERFACE)

- 앵커의 위치는 부모 구형의 **width**와 **height**를 분수(또는 퍼센트)로 정의한 것이다**. 0.0(0 %)**은 왼쪽 또는 하단에**, 0.5(50 %)**은 중간에**,** 그리고 **1.0(100 %)**은 오른쪽 또는 상단에 대응하고 있다**.** 그러나 앵커는 끝이나 중간에 제한되는 것은 아니다**.** 그들은 부모 사각형 내의 어떠한 지점에도 고정할 수 있다**.**



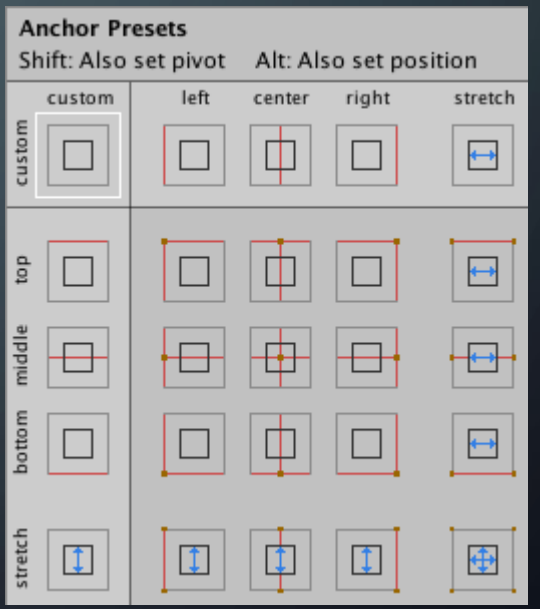
왼쪽 모서리를 부모 사각형의 왼쪽에서 특정 백분율만큼 떨어뜨려 고정하고, 오른쪽 모서리를 부모 사각형의 오른쪽 맨 끝에서 특정 백분율만큼 떨어뜨려 고정한 UI 요소.

2. UI(USER INTERFACE)

- 각각의 앵커를 개별적으로 드래그 할 수 있고 만약 함께 있으면, 그 중간을 클릭하고 드래그하여 함께 드래그 할 수 있다. 앵커를 Shift 키를 누른 상태로 드래그하면 사각형에 대응하는 코너는 앵커와 함께 움직인다.

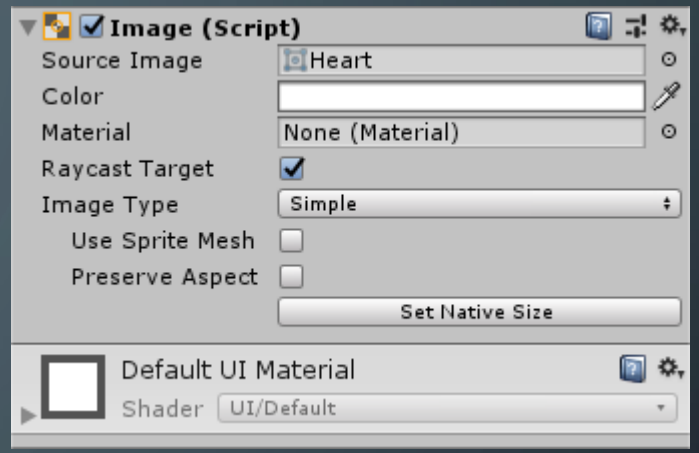
- **Anchor presets**

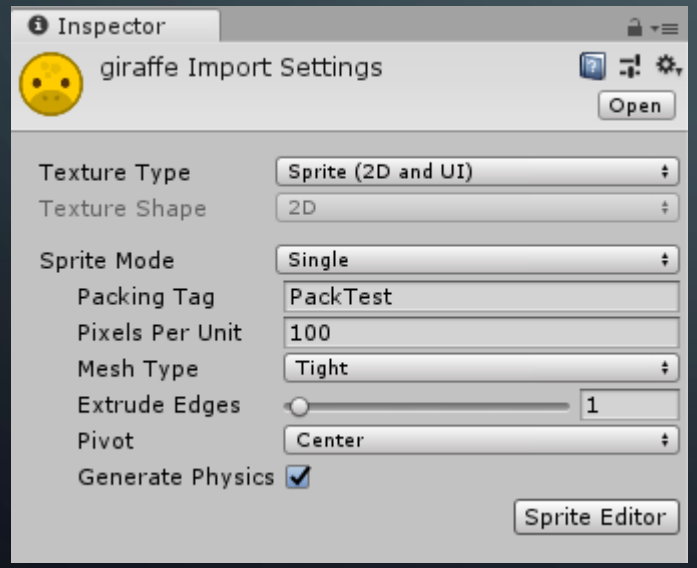
  - 인스펙터에서는 **Anchor Preset** 버튼이 **Rect Transform** 컴포넌트의 왼쪽 위의 모서리에 있다. 버튼을 클릭하면, 앵커 프리셋이 표시된다. 여기에서 바로 몇 가지 가장 일반적인 앵커 옵션 중 하나를 선택할 수 있고 **UI** 요소를 부모의 가장자리 또는 중간에 고정하거나 부모의 크기에 따라 늘릴 수 있다. 수평 및 수직 고정은 별도.
  - **Anchor Presets** 버튼은 선택되어 있는 것이 하나 있다면, 현재 선택되어 있는 프리셋 옵션을 표시한다. 수평 또는 수직 축에서 앵커가 프리셋 중 하나와 다른 위치에 설정되어 있는 경우, 사용자 지정 옵션이 표시된다.



2. UI(USER INTERFACE)

- **Image**
- **Source Image : 2DUI Sprite** 타입의 **Image**를 링크한다.
- **Color :** 지정한 컬러 값을 **Image**에 준다.
- **Material :** 특정한 셰이더를 추가하고 싶을 때 쓴다.
- **Image Type : Simple** – 원래 형식 그대로.
  - **Sliced - 9**등분한 곳 중 가운데만 크기만큼 늘려주고 싶을 때.
  - **Tiled - Image**를 타일배열로 크기만큼 채워준다.
  - **Filed -** 이미지를 수치에 따라 채워주고 싶을 때.

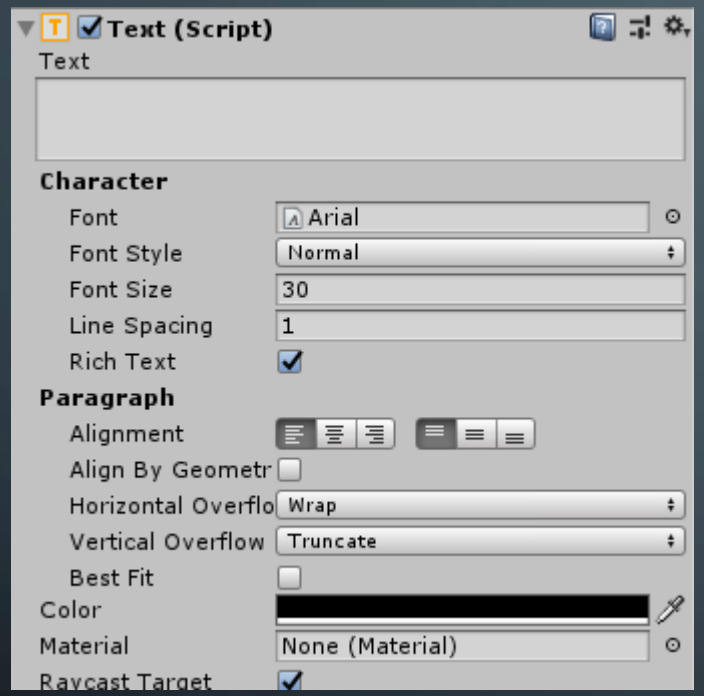




Image들은 저장된 형태에 따라 인식되는 Type이 있는데 Sprite형식으로 저장되어 있지 않을 것입니다. UNITY Sprite로 쓸 수 있는 형식으로 변경하는 것이 중요하다. Texture Type를 눌러서 보이는 것같이 Sprite(2D and UI)로 변경 시켜줘야 한다.

2. UI(USER INTERFACE)

- **TEXT**

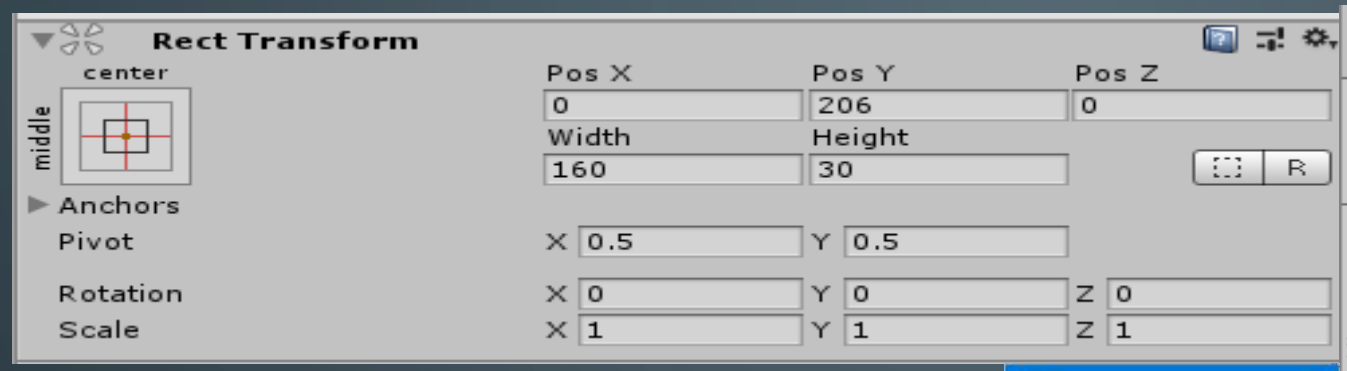


- **Text :** 표시할 문자열
- **Font :** 설정하고 싶은 문자열의 폰트, 추가폰트는 **Asset**에 있어야 한다.
- **FontStyle :** 폰트스타일을 지정한다.
- **Line Spacing :** 줄 간격.
- **Font Size : Font** 크기
- **Rich Text :** 서식이 있는 문자열 지원 **exhello** 굵은 글씨 **<color =#ff0000ff>**내가**</color>** 색깔변경.
- **Alignment :** 정렬 형태.
- **Horizontal Overflow :** 수평으로 넘어 갈 때, 어떻게 처리할 것인지 설정. **Wrap** – 다음 줄로, **Overflow** 넘어가게 둔다.
- **Vertical Overflow :** 라인을 넘어 갔을 때, 어떻게 처리할 것인지 설정.
- **Truncate -** 영역을 넘어가면 보이지 않게, **Overflow -** 영역을 넘어간 글자도 보이도록 할 때

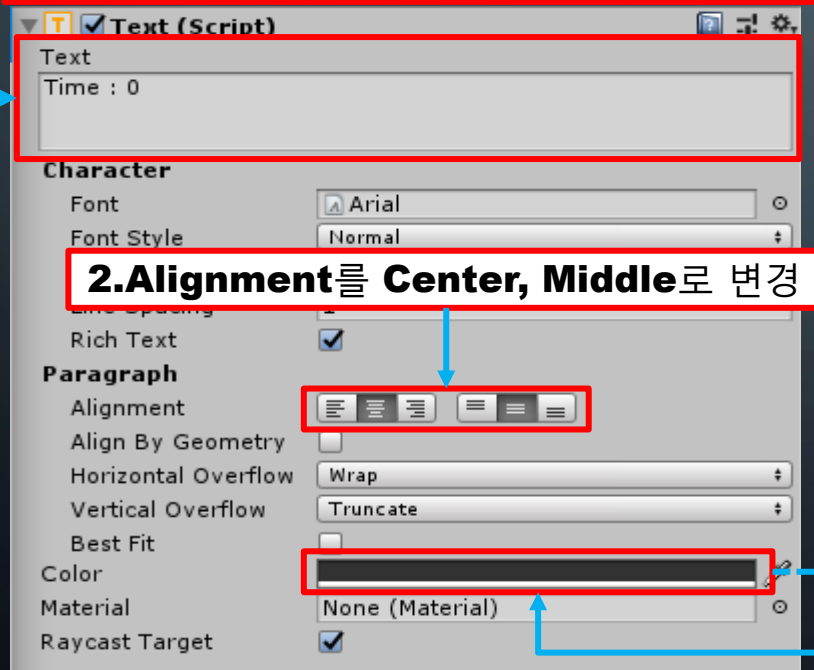
2. UI(USER INTERFACE)

TEXT 꾸미기

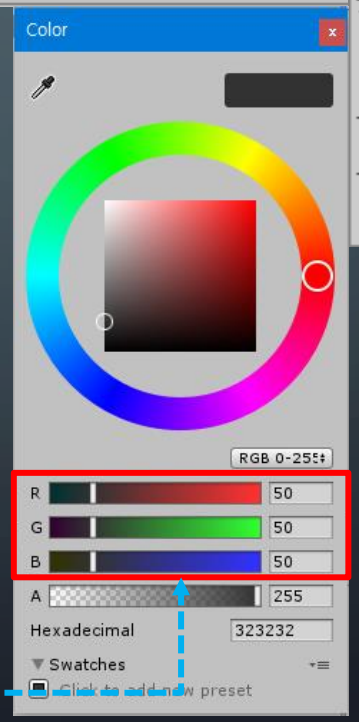
Anchor Preset을 이용해서 **Canvas**의 가운데 상단에 정렬 시켜보자.



1. Text Component의 **Text Filed** 내용을 **Time : 0**으로 변경



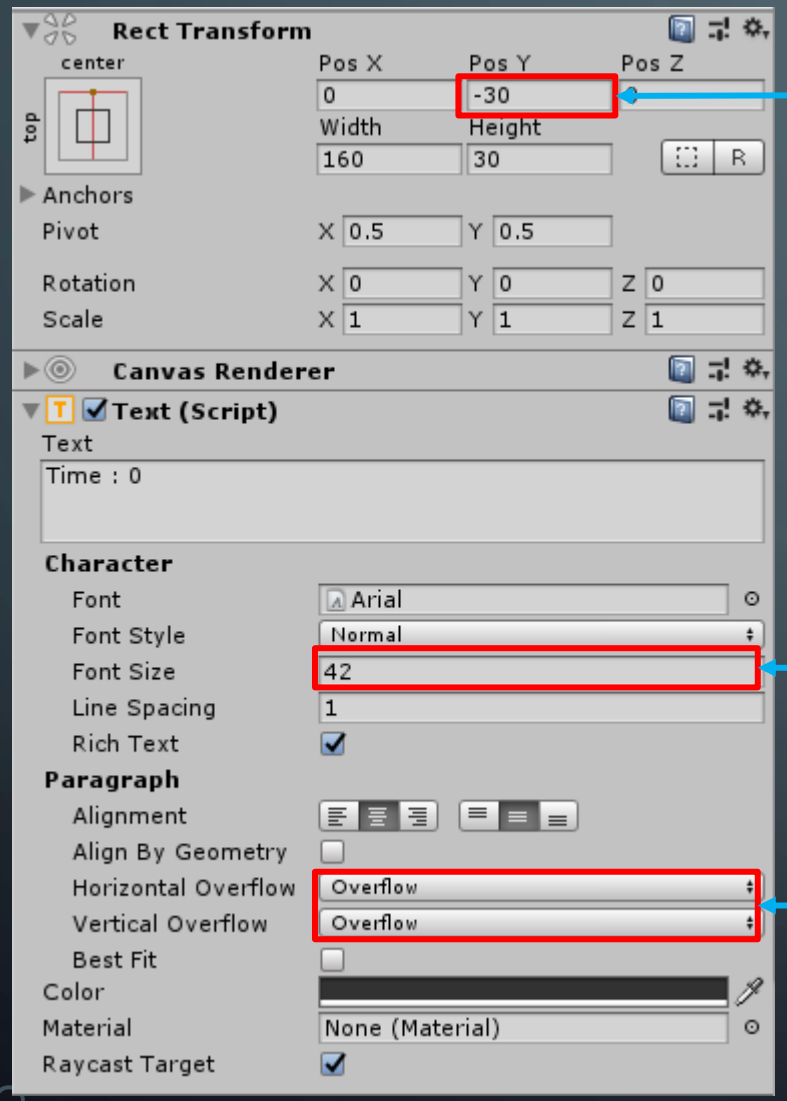
2.Alignment를 **Center, Middle**로 변경



3. Color Filed 클릭→**Font Color**를 하얀색**(255, 255, 255)**으로 변경

2. UI(USER INTERFACE)

Font Size와 위치를 조정하자**.**



1. Rect Transform Component의 **Pos Y**를 **-30**으로 변경

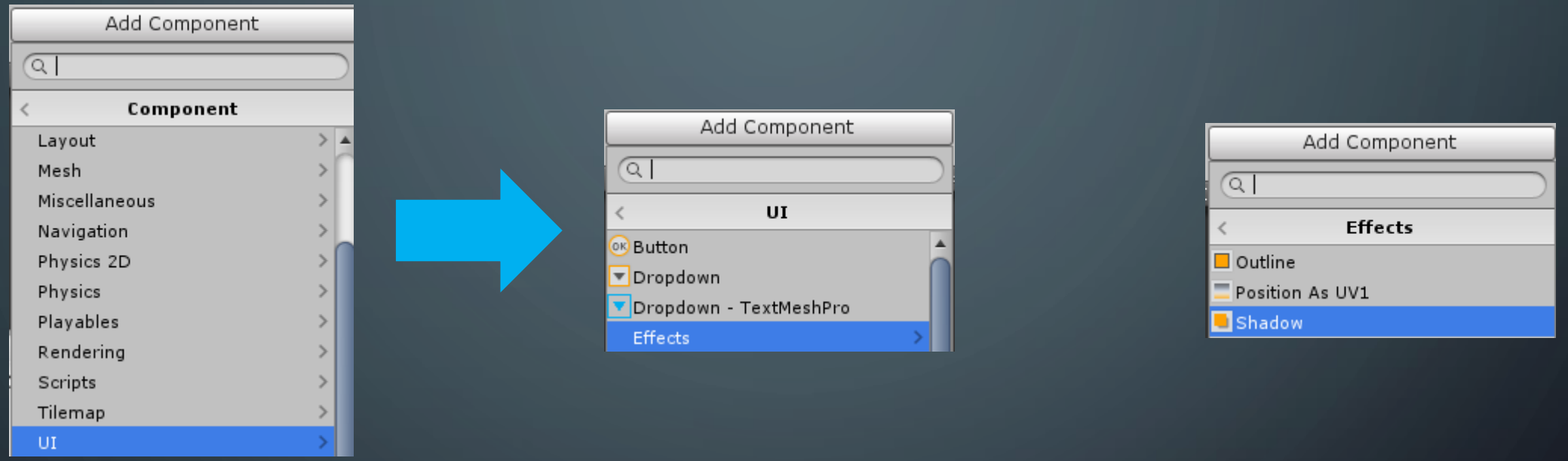
2. Font Size를 **42**로 변경

3. Horizontal Overflow와 **Vertical Overflow**를 **Overflow**로 변경

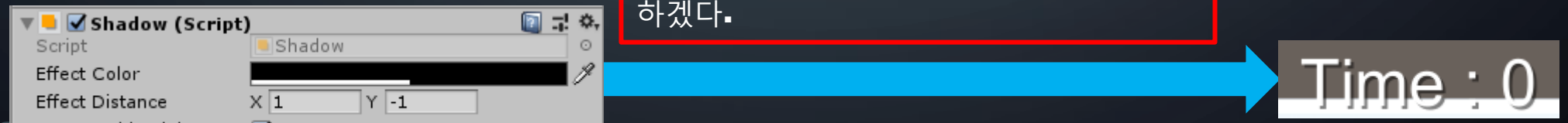
2. UI(USER INTERFACE)

Font에 **Effect** 효과를 추가하자.

- **Add Component → UI → Effect →** 원하는 **Effect(Shadow**를 예로**...)**



- **Effect**를 선택하고 속성을 변경해서 적용된 모습을 살펴보자.



Color값을 **(0, 0, 0, 128)**로 오른쪽으로 **1**, 아래 쪽으로 **-1** 각 방향으로 그림자를 생성하겠다.

2. UI(USER INTERFACE)

똑같은 방법으로 종료 후 다시 시작하는 **Text UI**를 만들어 보자**.**

2. UI(USER INTERFACE)

GameManager(싱글턴이 아님**) Component**를 작성해서 **GameEnd**와 **Time** 기록을 하고 **GameEnd**상황에서 **GameRestar**를 만들어 보자**.**