UNITY
-CAHPTER7-

SOUL SEEK

목차

C# 인터페이스를 이용한 DAMAGE 처리

1. C# INTERFACE

- 외부와 통신하는 공개 통로 이며 통로는 규격이다.
  →통로의 규격은 강제하지만 그 아래에 어떤 일이 일어날지는 결정하지 않는다.
  →**USB** 인터페이스를 생각해보면 **USB** 인터페이스를 가진 장비라면 **USB** 슬롯에 꽂아서 데이터를 주고 받을 수 있다. 하지만 **USB**로 연결된 장비 내부에서 일어나는 일은 **USB** 인터페이스 그 자체와는 상관없다.
- 인터페이스에 의해 선언된 **Method**는 인터페이스를 사용하는 객체에서 반드시 구현해 줘야 한다.

```
void OnTriggerEnter(Collider other)
{
    AmmoPack ammoPack = other.GetComponent<AmmoPack>();

    if(ammoPack != null)
        ammoPack.Use();

    HealthPack healthPack = other.GetComponent<HealthPack>();

    if (healthPack != null)
        healthPack.Use();
}
```

Item 습득을 위해 **Item Object**들과 충돌한 상황 에서 각 어떤 **Item Object**와 충돌했는지 파악해서 **Method**를 실행해줘야 한다.
→상속을 이용해서 간단하게 해결할 수 있을 것 같지만 이미 **Unity Class**는 **MonoBehaviour**를 상속하는 **Component**들이다. **C#**은 다중상속 중 여러 부모를 가지는 상속이 불가능 하기 때문에 이렇게 구현하게 된다.

```
void OnTriggerEnter(Collider other)
{
    IItem item = other.GetComponent<IItem>();
    //IItem item = other as IItem;

    if (item != null)
        item.Use();
}
```

```
public interface IItem
{
    // 입력으로 받는 target은 아이템 효과가 적용될 대상
    void Use(GameObject target);
}
```

```
public class Ammo : MonoBehaviour, IItem
{
    public int ammo = 30;

    public void Use(GameObject target)
    {
        Debug.Log("탄알이 증가했다.");
    }
}
```

```
public class Health : MonoBehaviour, IItem
{
    public int health = 50;

    public void Use(GameObject target)
    {
        Debug.Log("체력이 증가했다.");
    }
}
```

2. IDAMAGEABLE

- **IDamageable** 인터페이스를 상속하는 클래스는 **OnDamage() Method**를 반드시 구현해야 한다.
- →**Damage**를 받아서 **OnDamage**를 사용해야할 필요성이 있는 모든 **Componet**들은 **IDamageable**를 상속 시킨다.
- →**Player, Enemy(**기본**,** 보스**,** 새로운 타입**…**등등**)**각각의 데미지 계산이 다를 수 있기 때문에 각자의 **Damage** 구현부에서 **Damage**를 구현하게 한다.

```
// 데미지를 입을 수 있는 타입들이 공통적으로 가져야 하는 인터페이스
public interface IDamageable
{
    // 데미지를 입을 수 있는 타입들은 IDamageable을 상속하고 OnDamage 메서드를 반드시 구현해야 한다
    // OnDamage 메서드는 입력으로 데미지 크기(damage), 맞은 지점(hitPoint), 맞은 표면의 방향(hitNormal)을 받는다
    void OnDamage(float damage, Vector3 hitPoint, Vector3 hitNormal);
}
```

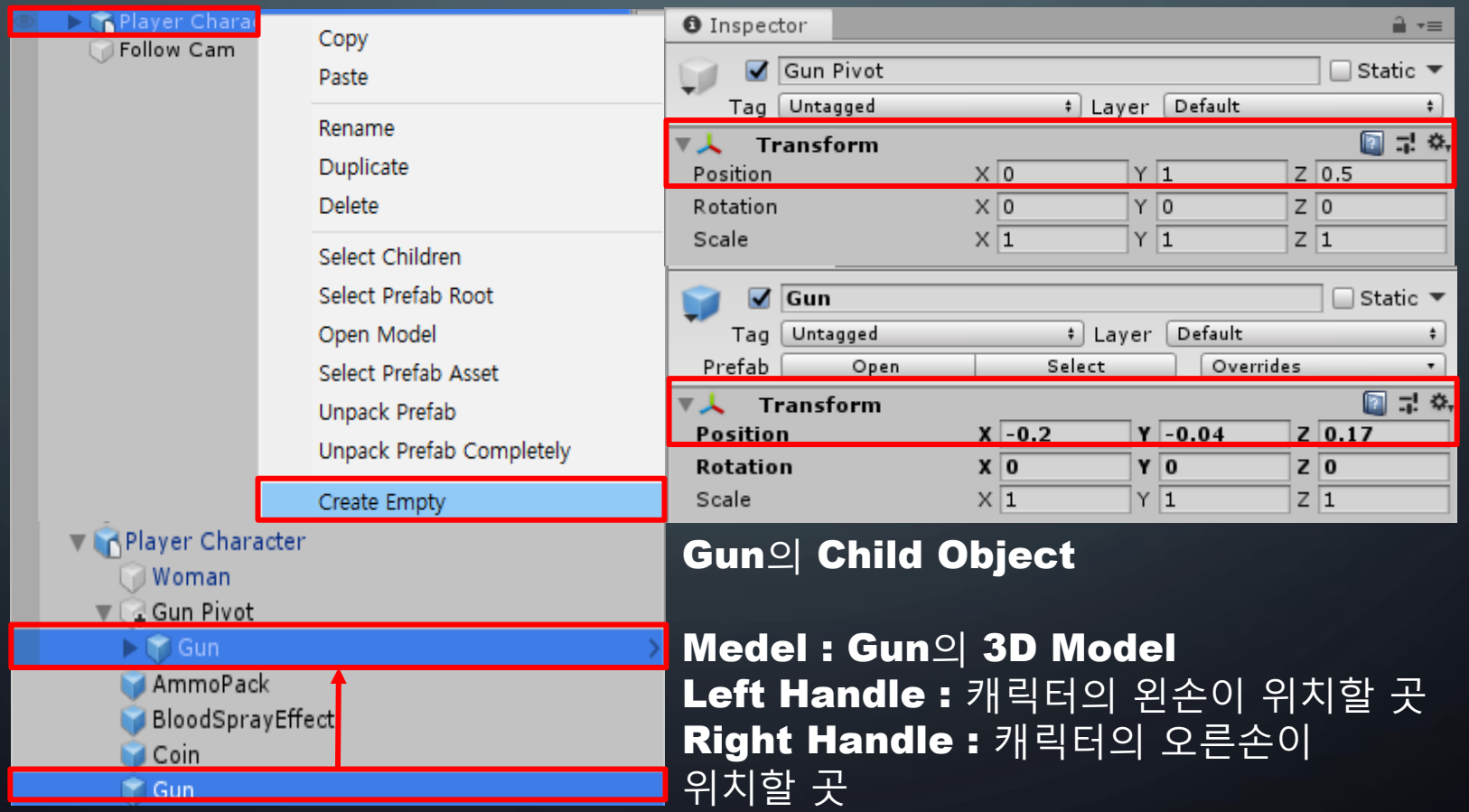
- **Damage :** 데미지 크기
- **hitPoint :** 공격당한 위치
- **hitNormal :** 공격당한 표면의 방향

→ **3**가지 매개변수들로 알 수 있는 건 받은 데미지 이외에 맞은 위치와 맞은 방향의 **Normal**을 이용해 이펙트의 위치와 이펙트가 보여지는 방향을 나타내기 위해서 이다.
→ 공격자에게 공격을 당하면 자신의 **Data**를 가지고 **Damage** 계산을 하면 될 것이다.
→ **Ex)** 속성의 상성이 필요하다면 속성을 설정하고 속성값도 매개변수로 받아서 자신의 속성과 상성을 계산하거나 무기의 속성으로 **Damage Type**을 적용하거나 공격타입으로 물리**,** 마법으로 상성을 맞춰주는 형식으로 사용할 수 있다.

3. GUN OBJECT

Gun GameObject 준비하기

- 하이어라키 창에서 **Player Character**를 마우스 오른쪽 클릭 **> Create Empty** 클릭**,** 생성된 자식 **GameObject**의 이름을 **Gun Pivot**으로**,** 위치를 **(0, 1, 0.5)** 변경한다**.**
- **Prefabs** 폴더의 **Gun Prefab**을 하이어라키 창의 **Gun Pivot**으로 **Drag & Drop,** 생성된 **Gun Game Object**의 위치를 **(-0.2, -0.44, 0.17)**로 변경



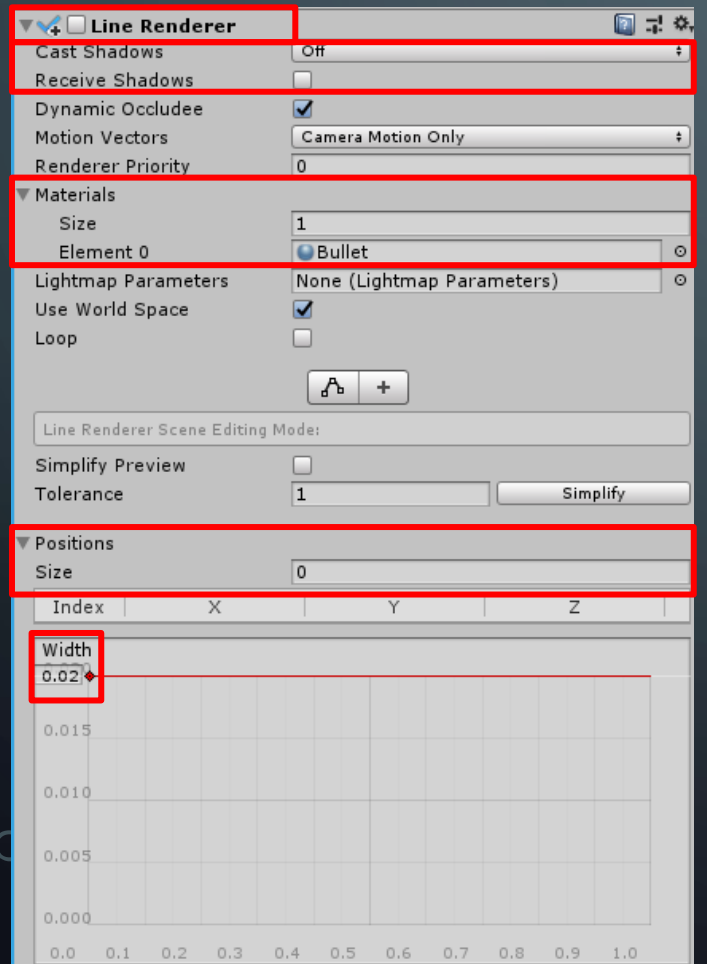
Gun의 **Child Object**

Medel : Gun의 **3D Model**
Left Handle : 캐릭터의 왼손이 위치할 곳
Right Handle : 캐릭터의 오른손이 위치할 곳

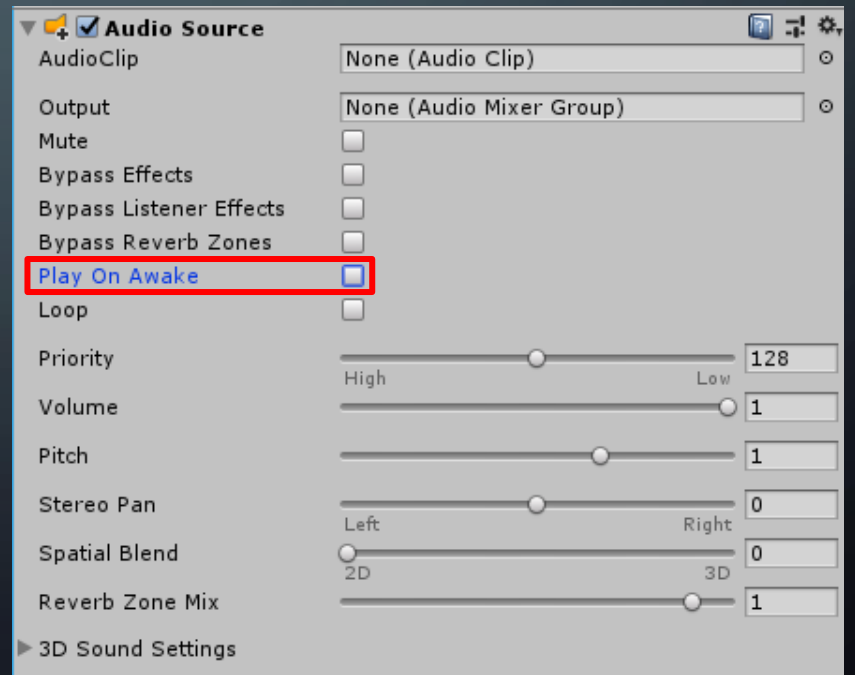
3. GUN OBJECT

총알이 발사되는 궤적표시와 Audio Source추가

- **Gun GameObject**에 **Line Renderer Component** 추가**(Add Component > Effect > Line Renderer), Line Renderer Component**를 체크 해제하여 비활성화
- **Cast Shadows**를 **off**로 변경**, Receive Shadows** 체크해제**, Materials** 탭 펼치기 **> Element 0**에 **Bullet Material** 할당**, Positions** 탭에서 **Size**를 **0**으로**, Width**를 **0.02**로 변경



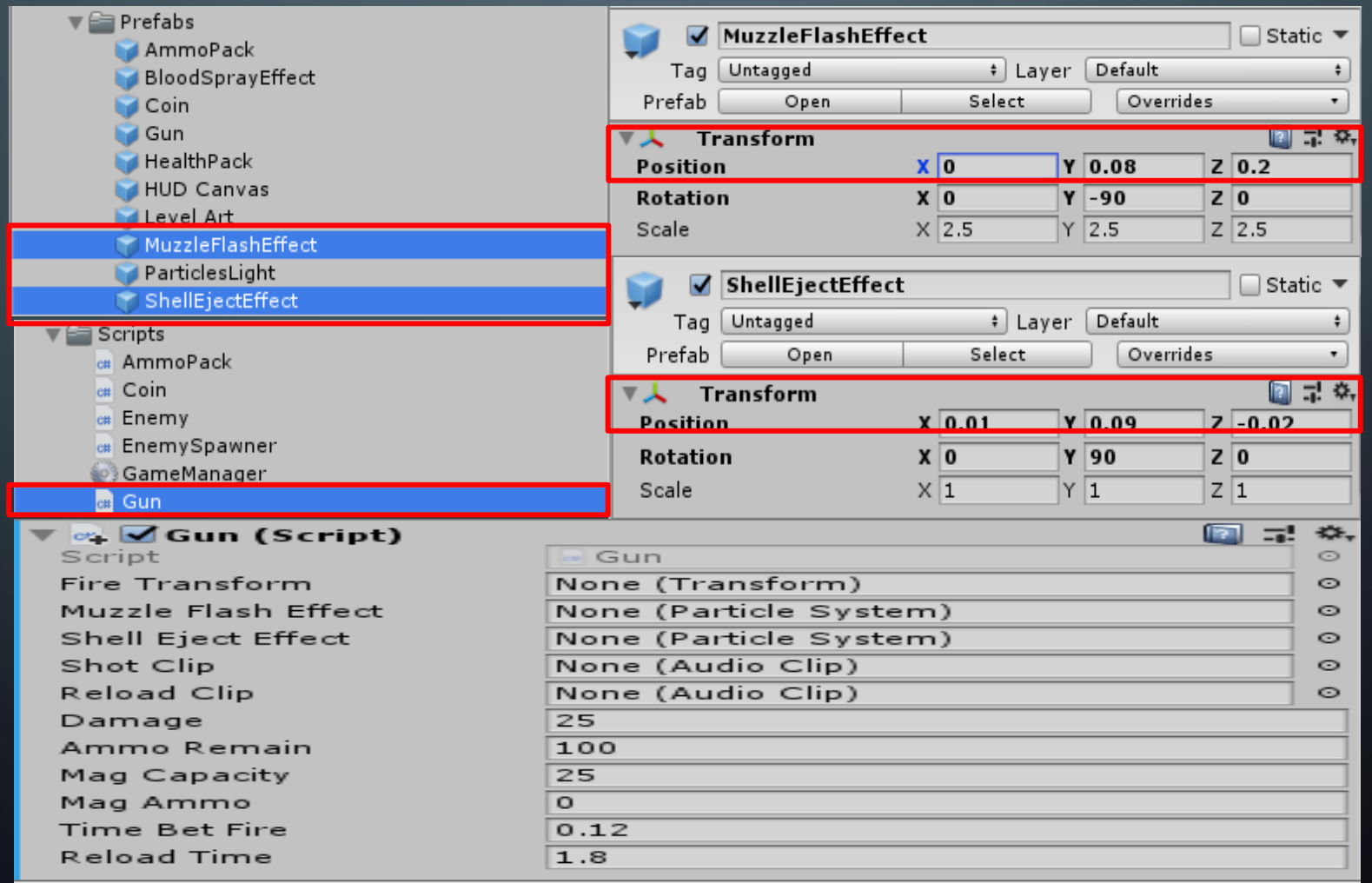
- **Gun GameObject**에 **Audio Source Component** 추가**(Add Component > Audio > Audio Source)**
- **Audio Source Component**의 **Play On Awake** 체크 해제



3. GUN OBJECT

Particle Effect, Gun Script 추가하기

- **Prefabs** 폴더의 **MuzzleFlashEffect**와 **ShellEjectEffect**를 하이어라키 창의 **Gun GameObject**로 **Drag & Drop, MuzzleFlashEffect**를 **(0, 0.08, 0.2), ShellEjectEffect**를 **(0.01, 0.09, -0.2)** 설정**Gun Script**로 추가.



3. GUN SCRIPT

Awake() Method는 사용할 컴포넌트를 가져오고, **OnEnable() Method**는 총의 상태를 초기화한다.
Fire() Method는 총을 발사하지만 실제 발사처리는 **Shot() Method**에서 처리한다.
ShotEffect는 **Method**를 발사 효과를 재생하고 탄알 궤적을 그리고 **Reload() Method**는 재장전을 시도한다.

```
// 총의 상태를 표현하는데 사용할 타입을 선언한다
public enum State
{
    READY,      // 발사 준비됨
    EMPTY,      // 탄창이 빔
    RELOADING   // 재장전 중
}

public State state { get; private set; } // 현재 총의 상태
```

총의 상태가 여러 상태가 되는 경우가 있기 때문에 **Enum**을 통해 총의 상태를 정의하고 상태를 비교해서 사용될 **Method**를 제어한다.

```
public Transform fireTransform; // 총알이 발사될 위치

public ParticleSystem muzzleFlashEffect; // 총구 화염 효과
public ParticleSystem shellEjectEffect; // 탄피 배출 효과

private LineRenderer bulletLineRenderer; // 총알 궤적을 그리기 위한 렌더러
```

fireTransform은 총구의 위치와 방향을 알려주는 **Component**로 **Gun GameObject**의 자식으로 있던 **Fire Position GameObject**의 **Transform Component**가 이곳에 할당된다.
bulletLineRenderer는 탄알 궤적을 그리기 위해 추가한 **Gun GameObject**의 **LineRenderer Component**이다. 나머지 두가지는 **Gun**을 발사 할 때 발생하는 **Effect**의 활성화에 사용한다.

3. GUN SCRIPT

```
private AudioSource gunAudioPlayer; // 총 소리 재생기
public AudioClip shotClip; // 발사 소리
public AudioClip reloadClip; // 재장전 소리
public float damage = 25; // 공격력
private float fireDistance = 50f; // 사정거리

public int ammoRemain = 100; // 남은 전체 탄약
public int magCapacity = 25; // 탄창 용량
public int magAmmo; // 현재 탄창에 남아있는 탄약
public float timeBetFire = 0.12f; // 총알 발사 간격
public float reloadTime = 1.8f; // 재장전 소요 시간
private float lastFireTime; // 총을 마지막으로 발사한 시점
```

gunAudioPlayer를 이용해서 상황에 맞는 **AudioClip**을 재생할 것이다.

Damage는 탄알 한 발의 공격력, **fireDistance**는 총의 사거리, **ammoRemain**은 남은 전체 탄알, **magCapacity**는 탄창의 용량이다. **magCapacity**보다 많은 탄알을 탄창에 넣을 수 없다.

timeBetFire는 발사 사이의 시간 간격, 이 값을 낮추면 연사력이 올라간다. **ReloadTime**은 재장전 시간, 재장전 동안 은 재장전을 취소하고 발사로 넘어갈 수 없는 상황을 연출, **lastFireTime**은 총을 마지막으로 발사한 시점, 연사를 구현할 때 사용.

```
private void Awake()
{
    // 사용할 컴포넌트들의 참조를 가져오기
    gunAudioPlayer = GetComponent<AudioSource>();
    bulletLineRenderer = GetComponent<LineRenderer>();

    // 사용할 점을 두 개로 변경
    bulletLineRenderer.positionCount = 2;
    // 라인 렌더러를 비활성화
    bulletLineRenderer.enabled = false;
}
private void OnEnable()
{
    // 현재 탄창을 가득 채우기
    magAmmo = magCapacity;
    // 총의 현재 상태를 총을 쏠 준비가 된 상태로 변경
    state = State.READY;
    // 마지막으로 총을 쏜 시점을 초기화
    lastFireTime = 0;
}
```

- **Awake() Metod**에서는 사용할 **Comp onent**들을 **GameObject**로 부터 가져 온다.
- 그 다음 **LineRenderer**가 사용할 점의 수를 **2**로 변경하고, **LineRenderer Component**를 미리 비활성화 한다.
  →**Inspector View**에서 활성화로 바뀌어 있을 수 있기 때문에 비활성화로 한번 더 설정해 준다.
- **OnEnable() Method**를 이용해서 **Componet** 가 활성화될 때마다 매번 실행되면서 총의 상태 와 기본 탄알을 초기화하는 처리를 구현한다.
- 탄창을 가득 채우고 총의 상태를 대기 상태로 바 꾸고 마지막 발사타임을 초기화 한다.

BULLET OBJECT & COROUTINE

1. COROUTINE

- 대기 시간을 가질 수 있는 **Method**
- **IEnumerator Type**을 반환해야 하며**,** 처리가 일시 대기할 곳에 **yield** 키워드를 명시해야 한다**.**

Ex) 방을 청소하는 **Method**

```
void CleaningHiouse()
{
    // A방청소
    // B방청소
    // C방청소
}
IEnumerator CleaningHouse()
{
    // A방청소
    yield return new WaitForSeconds(10f);
    // B방청소
    yield return new WaitForSeconds(20f);
    // C방청소
}
void LifeCycle()
{
    //설거지를 하자.

    //청소를 하자.
    StartCoroutine(CleaningHouse());

    //빨래를 하자.
}
```

모든 방의 청소가 다 끝날 때까지 아무것도 할 수 없다**.**

A방 청소를 하고 한동안 다른 일을 하다가 **10**초 후에 **B**방을 청소하고 **20**초 동안 다른 일을 한 후에 **C**방을 청소 한다**.**

청소를 모두 하고 빨래를 하는 것이 아니라 모든 방 청소를 진행하는 중간 중간 다른 일을 진행하고 있다**.**

1. COROUTINE

- **Unity**는 **Single Thread**기반의 엔진이기 때문에 **Multi Thread** 처리를 할 수 없다**.** 그러나 **Multi Thread**와 같이 비동기 처리해야 하는 로직이 필요하기에 이와 유사한 **Coroutine**사용하는 방법을 제시하였다**.**
- **Yield +** 구분을 조합하여 코드의 제어권한을 유니티의 메인 루틴으로 양보하는 연출을 한다**.**
- 상태감지에 따른 행동을 하라는 명령이 필요 할 뿐 항상 감지하여서 그 상태일땐 명령수행 중이니까 내릴 필요가 없어지므로 **Update** 구분에 쓸데없는 비교문을 줄일 수 있다**.**

Coroutine의 문법

- **Yield** 구문을 최초로 만난 시간 이후로 제한한 시간만큼 지나지 않았다면 계속 반환
  → **yield return new WaitForSeconds(**시간**);**
- 조건없이 **Yield**를 만나면 무조건 반환 ➔ **While** 구문과 함께 사용한다**.**
  → **yield return null;**

- **StartCoroutine() Method**는 두가지 방법으로 실행할 **Coroutine Method**를 입력 받는다**.**
  → **StartCoroutine(SomeCoroutine());** ➔ **Coroutine Method**를 실행한 반환 값**.**
  → **StartCoroutine(“SomeCoroutine”);** ➔ **Coroutine Method**의 이름

- **Coroutine Method**를 실행하면서 그 반환 값을 즉시 **StartCoroutine()**에 입력하는 방식은 다음과 같이 실행할 **Coroutine Method**에 입력 값을 전달할 수 있다**.**
  → **StartCoroutine(SomeCoroutine(100));**

- **Coroutine Method**의 이름을 **StartCoroutine()**에 문자열로 입력하고 **Coroutine**을 실행하는 방식은 나중에 **StopCoroutine() Method**를 사용해 실행중인 **Coroutine**을 도중에 종료할 수 있다**.**

1. COROUTINE

```
void Loading()
{
    // Data 로딩중

    Fade();

    // Scene 로딩중
}
void Fade()
{
    for(float f = 1f; f >= 0f; f -= 0.1f)
    {
        Color c = GetComponent<Renderer>().material.color;
        c.a = f;
        GetComponent<Renderer>().material.color = c;
    }
}
IEnumerator Fade()
{
    Color c = GetComponent<Renderer>().material.color;

    while (c.a > 0f)
    {
        c.a -= 0.1f;
        GetComponent<Renderer>().material.color = c;
        yield return null;
    }

    // Scene 로딩
}
void Loading()
{
    // Data 로딩중

    StartCoroutine(Fade());
}
```

Data를 로딩하고 **FadeIn, FadeOut**으로 전환하는 것을 보여주려고 할 때 **Fade()**가 다 끝났는지 알 수도 없기 때문에 **Scene**을 로딩하는 구문은 **Fade**에 사용한 구분들의 상태를 파악해서 진행하게 하는 제어문이 필요하게 되고 **Loading()** 구문이 업데이트 구문에 존재할 때 이 구문 아래로 진행이 불가능하다.

Fade Method를 **Coroutine**으로 만들고 **While** 구문을 진행 할 수 있는 조건이라면 한 프레임당 한 번만 반복 하면서 **yield return null**을 만나서 아래로 진행하지 않고 **MainFrame**구문으로 반환을 하게 된다. **While**문의 조건을 만족하지 못할 경우 아래 구문으로 진행해서 코드 블록이 끝난다.

2. SHOTEFFECT() METHOD

- **ShotEffect() Method**를 완성한다.

```
// 발사 이펙트와 소리를 재생하고 총알 궤적을 그린다
private IEnumerator ShotEffect(Vector3 hitPosition)
{
    // 총구 화연 효과 재생
    muzzleFlashEffect.Play();
    // 탄피 배출 효과 재생
    shellEjectEffect.Play();

    // 총격 소리 재생
    gunAudioPlayer.PlayOneShot(shotClip);
// 선의 시작점은 총구의 위치
bulletLineRenderer.SetPosition(0, fireTransform.position);
// 선의 끝점은 입력으로 들어온 충돌 위치
bulletLineRenderer.SetPosition(1, hitPosition);
// 라인 렌더러를 활성화하여 총알 궤적을 그린다
bulletLineRenderer.enabled = true;

// 0.03초 동안 잠시 처리를 대기
yield return new WaitForSeconds(0.03f);

// 라인 렌더러를 비활성화하여 총알 궤적을 지운다
bulletLineRenderer.enabled = false;
```

3. FIRE() METHOD

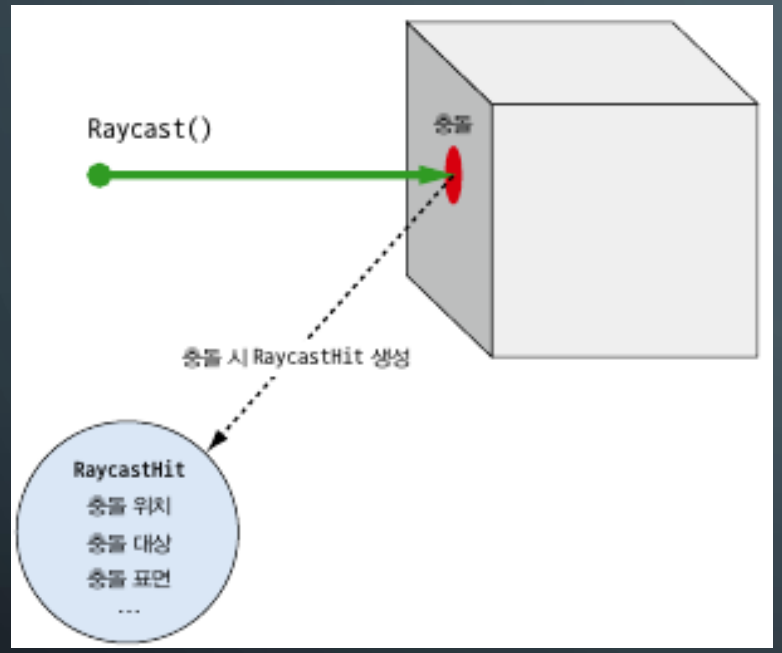
- **Fire() Method**를 완성한다.

```
public void Fire()
{
    // 현재 상태가 발사 가능한 상태
    // && 마지막 총 발사 시점에서 timeBetFire 이상의 시간이 지남
    if(state == State.READY && Time.time >= lastFireTime + timeBetFire)
    {
        // 마지막 총 발사 시점 갱신
        lastFireTime = Time.time;
        // 실제 발사 처리 실행
        Shot();
    }
}
```

- **IF** 구문에서 총의 현재 상태 **state**의 값이 총을 발사할 준비된 상태 **State.Ready**인지 검사하고 총을 발사하면서 발사 간격인 **timeBetFire**만큼 시간이 지나야 총을 다시 발사 할 수 있다는 조건을 걸어서 총을 발사할 수 있는지를 체크한다.
- 두가지 조건이 모두 만족하면 발사시간을 갱신하고 실제 발사 **Method**인 **Shot() Method**를 실행시킨다.

4. RAYCAST

- 사용자 눈에는 직접적으로 보이지 않는 방향을 가지는 선을 만들었을 때 다른 **Collider**와 충돌하는지 검사하는 처리에 사용하는 것을 **Ray**라고 한다**.**
- **Ray** 타입의 정보만 따로 표현할 수 있다**.**
- **RayCastHit** 이용해서 충돌정보를 알 수 있다**.**
  →**Ray**와 충돌한 **GameObject,** 충돌한 위치**,** 충돌한 표면의 방향 등을 알 수 있다**.**



- 탄도학 같은 물리 검증이 들어가 있는 **FPS, TPS**등이 아닌 일반 **FPS, TPS** 슈터 게임은 대부분 **RayCast**를 이용해 총을 구현한다**.**
- 총구에 **Ray**를 발사해서 다른 오브젝트와 충동하는지 검사하고 충돌한 오브젝트를 총에 맞은 것으로 처리하는 방식**.**
- 충돌위치와 표면으로 다양한 이펙트를 그 위치와 방향으로 연출하는데 활용할 수 있다**.**

5. SHOT() METHOD

```
private void Shot()
{
    //Raycast에 의한 충돌 정보를 저장하는 컨테이너
    RaycastHit hit;

    //탄알이 맞은 곳을 저장할 변수
    Vector3 hitPosition = Vector3.zero;
```

- **hit**는 **Raycast**의 결과를 저장할 변수
- **hitPosition**은 탄알이 충돌한 위치를 저장한다.
  →**Shot() Method**에서 계산된 **hitPosition**은 나중에 **ShotEffect() Method**의 입력으로 사용.

- **Physics.Raycast() Method**는 **Ray**를 쏴서 **Ray**와 충돌한 **Collider**가 있는지 검사한다.
  →충돌했다면 **true,** 실패했다면 **false,** 성공 시 **RaycastHit** 정보를 담아서 반환.
  →이를 이용해서 총을 쏘고, 총에 맞는 오브젝트가 있는지 검사한다.
  →총구위치, 총구의 앞쪽방향, 반환 받을 충돌 정보, 사거리

```
//Raycast(시작 지점, 방향, 충돌 정보 컨테이너, 사정거리)
if(Physics.Raycast(fireTransform.position, fireTransform.forward, out hit, fireDistance))
{
    //Ray가 어떤 물체와 충돌한 경우

    //충돌한 상대방으로 부터 IDamageable Component를 가져온다.
    IDamageable target = hit.collider.GetComponent<IDamageable>();

    //상대방으로 부터 IDamageable Component를 가져오는데 성공했다면
    if(target != null)
    {
        //상대방의 OnDamage 함수를 실행시켜 상대방에 Damage주기
        target.OnDamage(damage, hit.point, hit.normal);
    }

    //Ray가 충돌한 위치 저장
    hitPosition = hit.point;
}
else
{
    //Ray가 다른 물체와 충돌하지 않았다면
    //탄알이 최대 사정거리까지 날아갔을 때의 위치를 충돌 위치로 사용
    hitPosition = fireTransform.position + fireTransform.forward * fireDistance;
}
```

```
//발사 이펙트 재생 시작
StartCoroutine(ShotEffect(hitPosition));

//남은 탄알 수를 차감
magAmmo--;

if(magAmmo <= 0)
{
    //탄창에 남은 탄알이 없다면 총의 현재 상태를 Empty로 갱신
    state = State.EMPTY;
}
```

5. RELOAD() METHOD

- 재장전 실행에 성공하면 **true,** 재장전을 할 수 없는 상태면 **false**를 반환한다**.**

```
public bool Reload()
{
    if(state == State.RELOADING || ammoRemain <= 0 || magAmmo > magCapacity)
    {
        // 이미 재장전 중이거나 남은 탄알이 없거나
        // 탄창에 탄알이 이미 가득한 경우 재장전할 수 없다.
        return false;
    }

    //재장전 처리 시작
    StartCoroutine(ReloadRoutine());
    return true;
}
```

- **state == state.Reloading :** 이미 재장전을 하고 있는 중
- **ammoRemain <= 0 :** 재장전에 사용할 남은 탄알이 없음
- **magAmmo >= magCapacity :** 탄창에 탄알이 이미 가득 차 있음

- **If**문으로 조건들을 검색하여 하나라도 만족하면 재장전을 실행하지 않고 **false**를 반환하거나 **Reload() Method**를 즉시종료**,** 재장전을 할 수 있다면 **ReloadRoutine()**을 통해 **Coroutine**으로 재장전 상태 처리를 전환하고 **true**를 반환

5. RELOADROUTINE() METHOD

- 재장전 실행에 성공하면 **true,** 재장전을 할 수 없는 상태면 **false**를 반환한다.

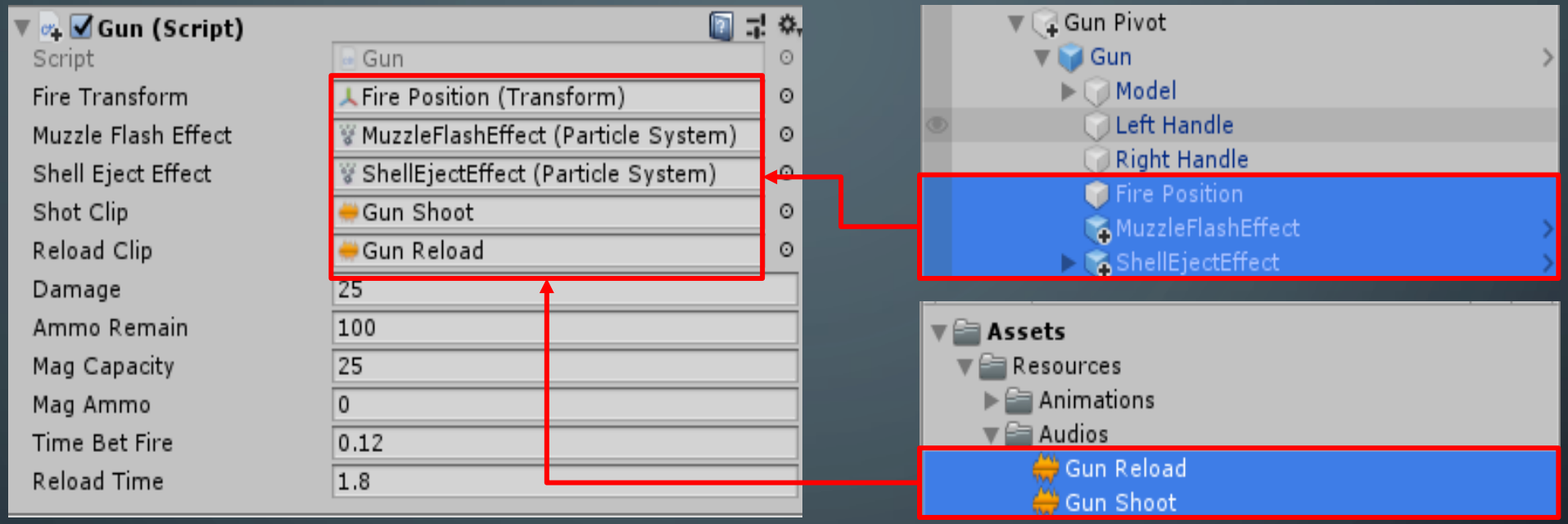
```
private IEnumerator ReloadRoutine()
{
    // 현재 상태를 재장전 중 상태로 전환
    state = State.RELOADING;
// 재장전 소리 재생
gunAudioPlayer.PlayOneShot(reloadClip);

// 재장전 소요 시간 만큼 처리를 쉬기
yield return new WaitForSeconds(reloadTime);
// 탄창에 채울 탄알을 계산
int ammoToFill = magCapacity - magAmmo;

// 탄창에 채워야할 탄알이 남은 탄알보다 많다면
// 채워야 할 탄알 수를 남은 탄알 수에 맞춰 줄임
if(ammoRemain < ammoToFill)
{
    ammoToFill = ammoRemain;
}
// 탄창을 채움
magAmmo += ammoToFill;
// 남은 탄알에서 탄창에 채운만큼 탄알을 뺌
ammoRemain -= ammoToFill;

// 총의 현재 상태를 발사 준비된 상태로 변경
state = State.READY;
```

- 현재 상태를 재장전 중인 상태로 변경.
- 오디오 소스로 재장전 오디오 클립을 **1**회 재생
- 재장전 시간 동안 반환 처리
  →여러 **Item**을 이용한 재장전 시간을 처리한다.
- 대기 시간동안 **state**의 값이 **Reloading**으로 고정되기 때문에 **Fire()**나 **Reload()**가 실행되어도 탄알이 발사되거나 중복 재장전이 실행되지 않는다.
- 대기 시간이 끝나면 탄창에 채워 넣어야 할 탄알 수 **ammoToFill**을 계산한다. 채워 넣을 탄알 수는 탄창의 최대 용량에서 탄창에 남아 있는 탄알수를 빼서 구한다.
- 남은 전체 탄알이 탄창에 채워 넣어야 하는 탄알보다 적다면 채워 넣을 탄알을 남은 탄알에 맞춘다. 가지고 있는 탄알보다 더 많은 탄알을 장전할 수 없기 때문이다.
- 채워 넣을 탄알 **ammoToFill**의 값을 계산한 다음에는 그만큼 탄창에 탄알을 집어넣는다.
- 탄창에 넣은 탄알 만큼 전체 탄알을 감소시키고 총의 현재 상태를 발사 준비된 상태로 변경하여 총이 다음 상태로 전환될 수 있게 한다.

PLAYERSHOOTER – FK / IK를 이용한 GUN 연동

1. GUN COMPONENT SETTING

Gun Component Filed 채우기

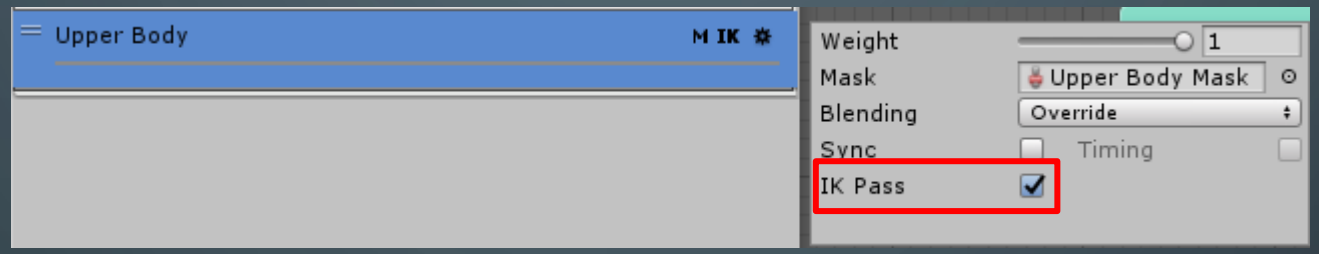


Audio Clip을 할당할 때는 **Shot Clip** 필드와 **Reload Clip** 필드 옆의 선택 버튼을 클릭해서 **Audio Clip** 선택창을 띄우고 할당할 **Audio Clip**을 선택하면 된다.
아직 총을 쏘는 슈터를 만들지 않았기 때문에 총을 쏘거나 재장전 할 수 없다.

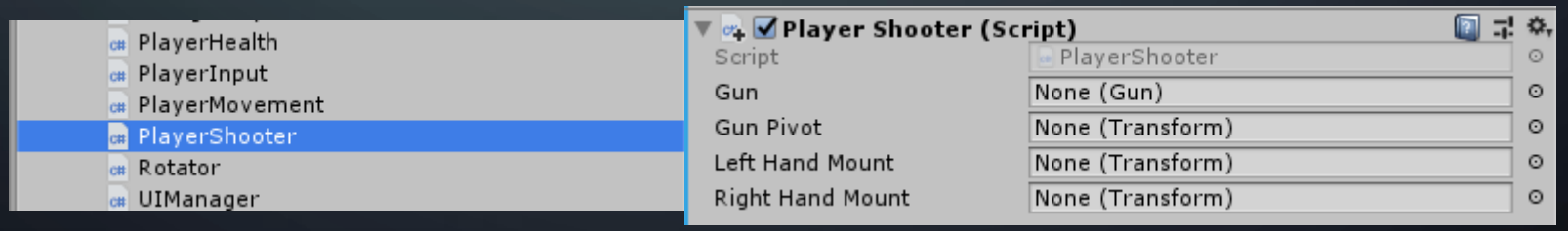
2. MAKE SHOOTER

PlayerShooter Script의 구현내용은
→플레이어 입력에 따라 총을 쏘거나 재장전 한다.
→플레이어 캐릭터의 손이 항상 총의 손잡이에 위치하도록 한다.
→상태에 따라 애니메이션이 변하고 있기 때문에 캐릭터 손의 위치가 항상 총의 손잡이에 위치하려면 **Animator**의 **IK**를 사용해야 한다.

IK를 사용하기 위해서는 **Animator Controller**에서 **IK Pass** 설정이 켜져 있어야 한다.



Animator Component가 **IK** 정보를 갱신할 때마다 **OnAnimatorIK** 메시지가 발생.
→**Script**에서 **IK**정보가 갱신될 때마다 자동 실행되는 **OnAnimatorIK() Method**를 구현하면 **IK**를 어떻게 사용할지 코드로 작성 할 수 있다.



2. MAKE SHOOTER

Gun Component를 할당 받을 변수와 **IK**갱신에 사용할 변수들을 선언.

```
public Gun gun; // 사용할 총
public Transform gunPivot; // 총 배치의 기준점
public Transform leftHandMount; // 총의 왼쪽 손잡이, 왼손이 위치할 지점
public Transform rightHandMount; // 총의 오른쪽 손잡이, 오른손이 위치할 지점
```

Player의 다른 **Component**에 대한 변수 선언

```
private PlayerInput playerInput; // 플레이어의 입력
private Animator playerAnimator; // 애니메이터 컴포넌트
```

사용 할 **Animator Component**와 **PlayerInput Component**에 대한 참조를 가져온다.

```
private void Start()
{
    // 사용할 컴포넌트들을 가져오기
    playerInput = GetComponent<PlayerInput>();
    playerAnimator = GetComponent<Animator>();
}
```

Component가 활성화와 비활성화될 때 자동으로 실행된다.

```
private void OnEnable()
{
    // 슈터가 활성화될 때 총도 함께 활성화
    gun.gameObject.SetActive(true);
}
```

```
private void OnDisable()
{
    // 슈터가 비활성화될 때 총도 함께 비활성화
    gun.gameObject.SetActive(false);
}
```

2. MAKE SHOOTER

PlayerInput을 통해 발사 상태를 감지하면 **gun.Fire()**를 실행.
재장전 입력을 감지하면 **gun.Reload**를 실행하여 총을 재장전 한다.

```
private void Update()
{
    // 입력을 감지하고 총을 발사하거나 재장전
    if(playerInput.fire)
    {
        // 발사 입력 감지 시 총 발사
        gun.Fire();
    }
    else if(playerInput.reload)
    {
        // 재장전 입력 감지 시 재장전
        if(gun.Reload())
        {
            //재장전 성공 시에만 재장전 애니메이션 재생
            playerAnimator.SetTrigger("Reload");
        }
    }
    //남은 탄알 UI 갱신
    UpdateUI();
```

```
private void UpdateUI()
{
    if (gun != null && UIManager.instance != null)
    {
        // UI 매니저의 탄약 텍스트에 탄창의 탄약과 남은 전체 탄약을 표시
        UIManager.instance.UpdateAmmoText(gun.magAmmo, gun.ammoRemain);
    }
}
```

2. MAKE SHOOTER

OnAnimatorIK() Method

- 총을 상체와 함께 흔들기
- 캐릭터의 양손을 총의 양쪽 손잡이에 위치시키기

```
private void OnAnimatorIK(int layerIndex)
{
    // 총의 기준점 gunPivot을 3D 모델의 오른쪽 팔꿈치 위치로 이동
    gunPivot.position = playerAnimator.GetIKHintPosition(AvatarIKHint.RightElbow);

    // IK를 사용하여 왼손의 위치와 회전을 총의 왼쪽 손잡이에 맞춘다.
    playerAnimator.SetIKPositionWeight(AvatarIKGoal.LeftHand, 1.0f);
    playerAnimator.SetIKRotationWeight(AvatarIKGoal.LeftHand, 1.0f);

    playerAnimator.SetIKPosition(AvatarIKGoal.LeftHand, leftHandMount.position);
    playerAnimator.SetIKRotation(AvatarIKGoal.LeftHand, leftHandMount.rotation);

    // IK를 사용하여 오른손의 위치와 회전을 총의 오른쪽 손잡이에 맞춘다.
    playerAnimator.SetIKPositionWeight(AvatarIKGoal.RightHand, 1.0f);
    playerAnimator.SetIKRotationWeight(AvatarIKGoal.RightHand, 1.0f);

    playerAnimator.SetIKPosition(AvatarIKGoal.RightHand, rightHandMount.position);
    playerAnimator.SetIKRotation(AvatarIKGoal.RightHand, rightHandMount.rotation);
}
```

2. MAKE SHOOTER

오른쪽 팔꿈치를 위치를 찾아 **gunPivot**의 위치로 사용**.**

```
// 총의 기준점 gunPivot을 3D 모델의 오른쪽 팔꿈치 위치로 이동
gunPivot.position = playerAnimator.GetIKHintPosition(AvatarIKHint.RightElbow);
```

Animator Component의 **GetIKHintPosition() Method**는 **AvatarIKHint Type**으로 부위를 입력 받아 해당 부위의 현재 위치를 가져온다**.**

AvatarIKHint Type
AvatarIKHint.LeftIbow, AvatarIKHInt.RightIbow, AvatarIKHint.LeftKnee, AvatarIKHint.RightKnee

캐릭터의 왼손의 위치와 회전을 **leftHandMount**의 위치와 회전으로 변경**.**
→왼손**IK**에 대한 위치와 회전 가중치를 **1.0(100%)**으로 변경
→**IK**대상의 가중치를 설정할 때 위치는 **SetIKPositionWeight(),** 회전은 **SetIKRotationWeight()**를 사용
→왼손**IK**의 목표위치와 회전을 **leftHandMount**의 위치와 회전으로 지정**. SetIKPosition(), SetIKRotation()** 사용

```
// IK를 사용하여 왼손의 위치와 회전을 총의 왼쪽 손잡이에 맞춘다.
playerAnimator.SetIKPositionWeight(AvatarIKGoal.LeftHand, 1.0f);
playerAnimator.SetIKRotationWeight(AvatarIKGoal.LeftHand, 1.0f);

playerAnimator.SetIKPosition(AvatarIKGoal.LeftHand, leftHandMount.position);
playerAnimator.SetIKRotation(AvatarIKGoal.LeftHand, leftHandMount.rotation);
```

AvatarIKGoal Type
AvatarIKGoal.LeftHand, AvatarIKGoal.RightHand, AvatarIKLeftFoot, AvatarIKRightFoot

```
// IK를 사용하여 오른손의 위치와 회전을 총의 오른쪽 손잡이에 맞춘다.
playerAnimator.SetIKPositionWeight(AvatarIKGoal.RightHand, 1.0f);
playerAnimator.SetIKRotationWeight(AvatarIKGoal.RightHand, 1.0f);

playerAnimator.SetIKPosition(AvatarIKGoal.RightHand, rightHandMount.position);
playerAnimator.SetIKRotation(AvatarIKGoal.RightHand, rightHandMount.rotation);
```

2. MAKE SHOOTER

Field 채우기