UNITY
-CAHPTER7-

SOUL SEEK

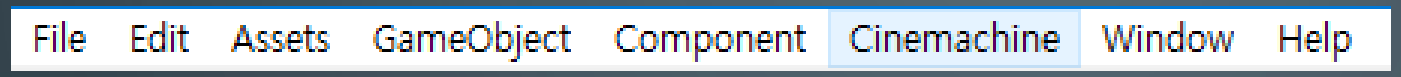
목차

LINGHTING 설정

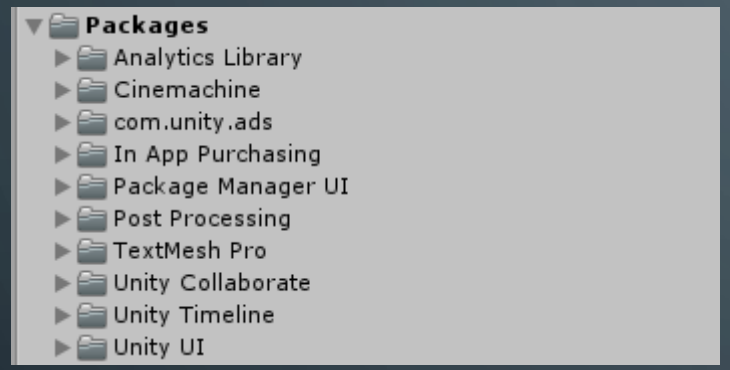
1. 프로젝트 구성

Standard Packages

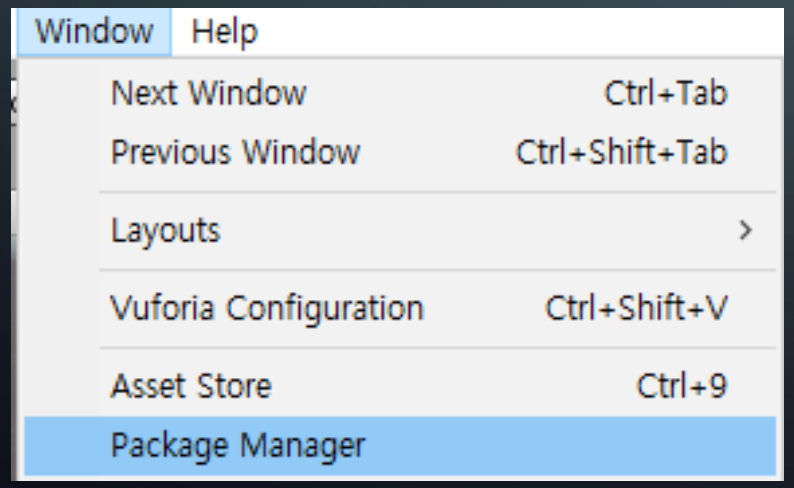
Unity에서 사용자의 편의를 위해 제공하고 있는 **Package**들이다 필요여부에 따라 **Package Manager**를 통해 **Import**해서 사용할 수 있다.

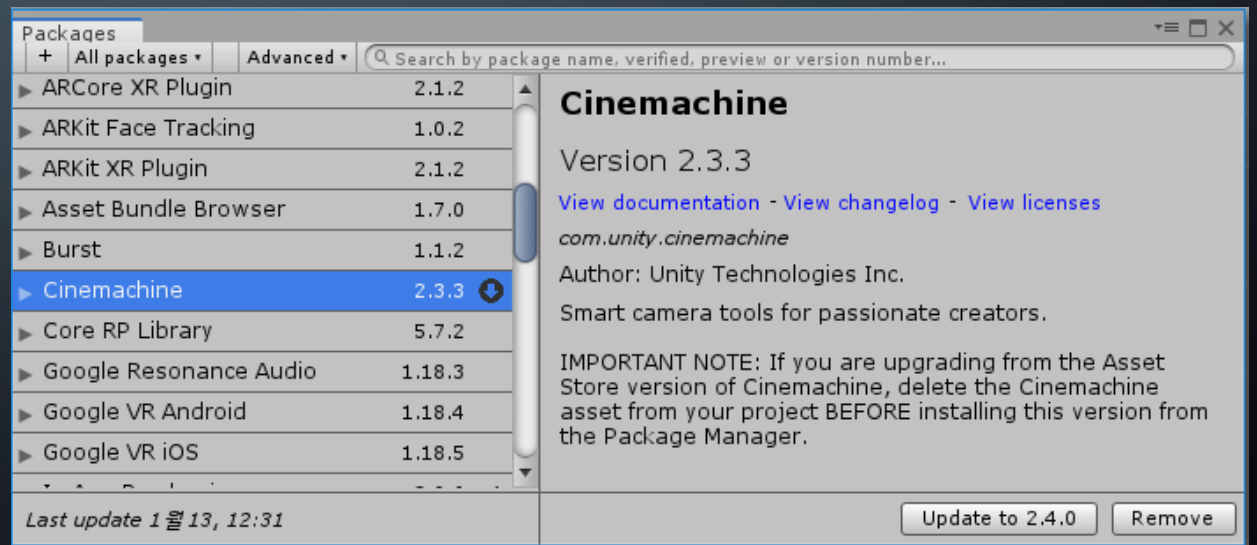


Assets 폴더 아래에 **Packages** 폴더가 **Standard Packages**들이 저장되는 폴더이다.



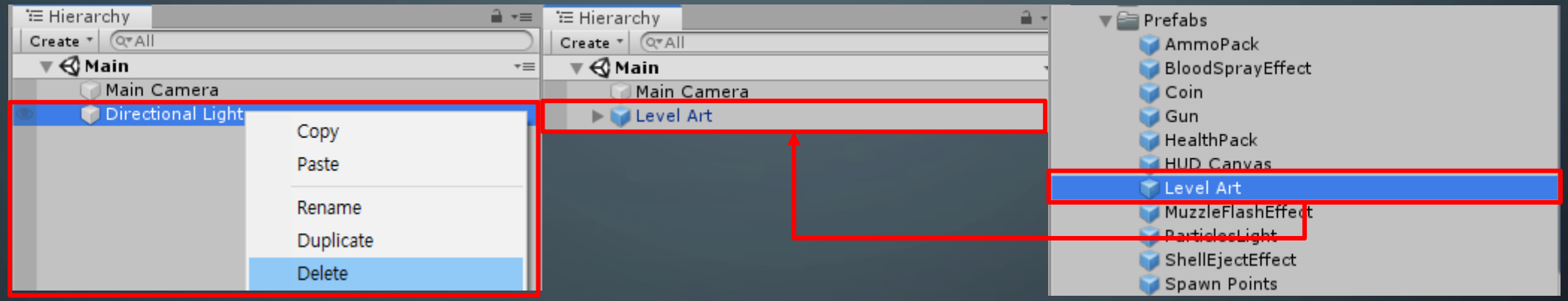
Standard Packages는 **Window > Package Manager**에서 나타나는 **Manager** 툴을 이용해 원하는 **Package**들을 적용할 수 있다.



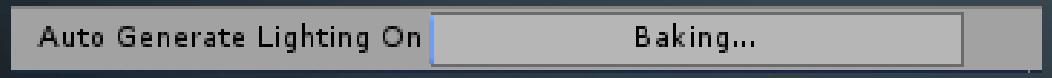


2. LIGHT MAP

- **MainScene**을 만들고 **Level Art**를 가져와서 기본 지형을 구성하자.
  →**Level** 구성요소(**Level**이나 난이도, 밸런스에 영향을 미치는 요소)을 구축하는 것이다.
- 기본 설치된 **Directional Light**는 삭제하자.
  →**Level Art**에 **LightMap**과 **Light**가 설치 되었기 때문에 삭제했다.
  →해당 **Prefab**을 적용하면 이미 **LightMap**이 적용되어 있기 때문에 자동 **Baking**을 시도하기 때문에 **Load** 시간이 걸릴 것이다.



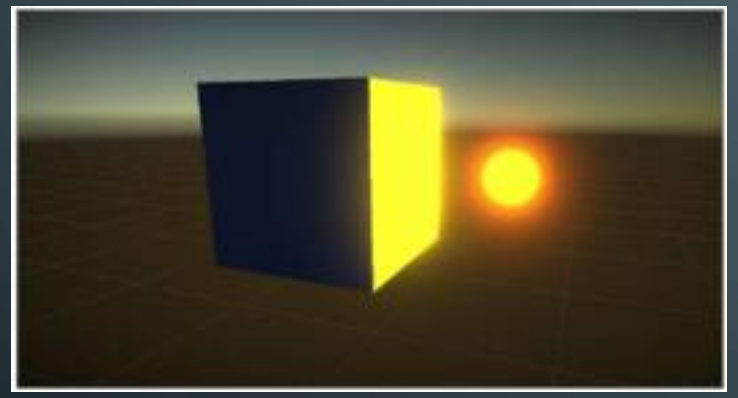
작은 변화에도 매번 **Baking**하기 때문에 당분간은 수동으로 사용해야 작업에 지장이 없다.



첫 **Baking**에 시간이 꽤 걸린다.

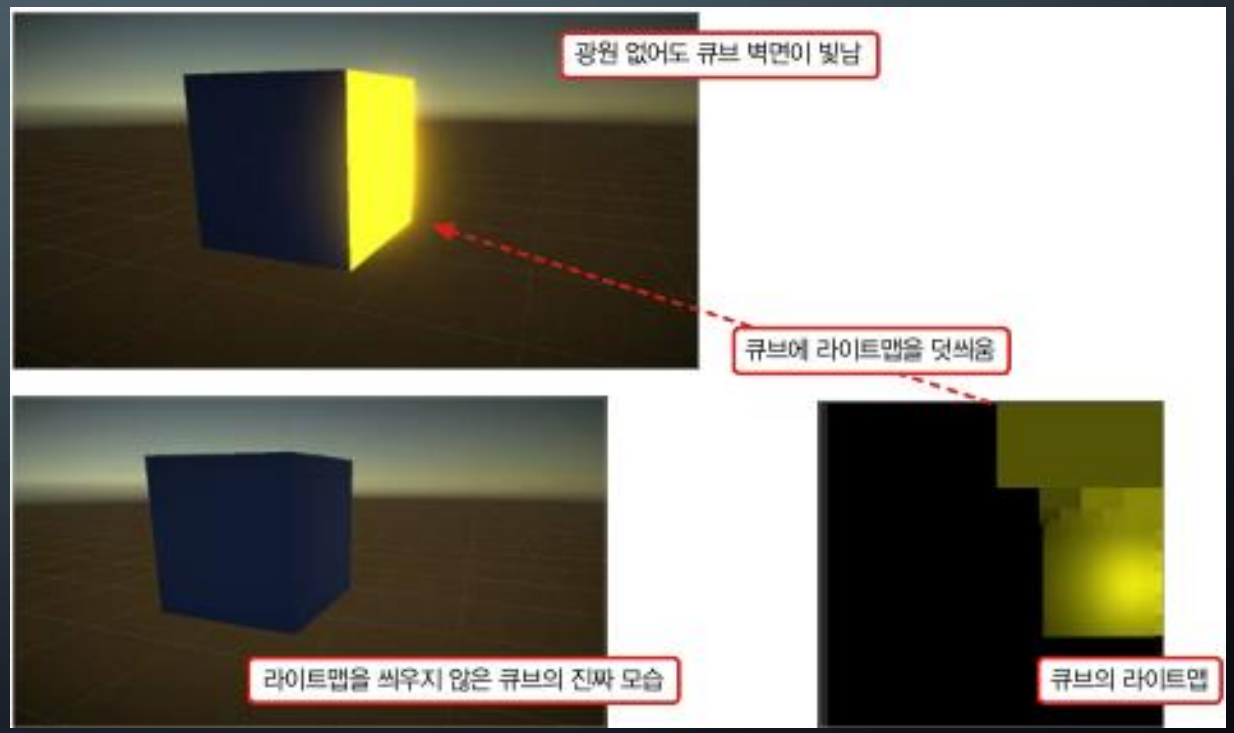
2. LIGHT MAP

- **Unity**는 **Lighting Data Asset**을 사용하여 **Lighting Effect**의 실시간 연산 량을 줄이며, **Scene**에 변화가 감지 때마다 매번 새로운 **Lighting Asset**을 생성**(Baking)**한다.
  →**Lighting** 연산이 비싸기 때문에 미리미리 조금씩 자주 해놓는게 연산 량을 줄이는 것이기 때문.
  →**Lighting Data Asset**에 포함된 주요 **Data**중 하나가 **Light Map**이다.
- **Light Map :** 오브젝트가 빛을 받았을 때 어떻게 보여질지 미리 그려진 **Texture.**
  →물체의 표면위에 데칼을 입히는 것으로 이해할 수 있다.
  →실시간 광원이 없이 빛을 내는 것처럼 보이는 **Light Object**를 배치하고 이를 실시간 광원이 적용되는 것처럼 보이게 한다.



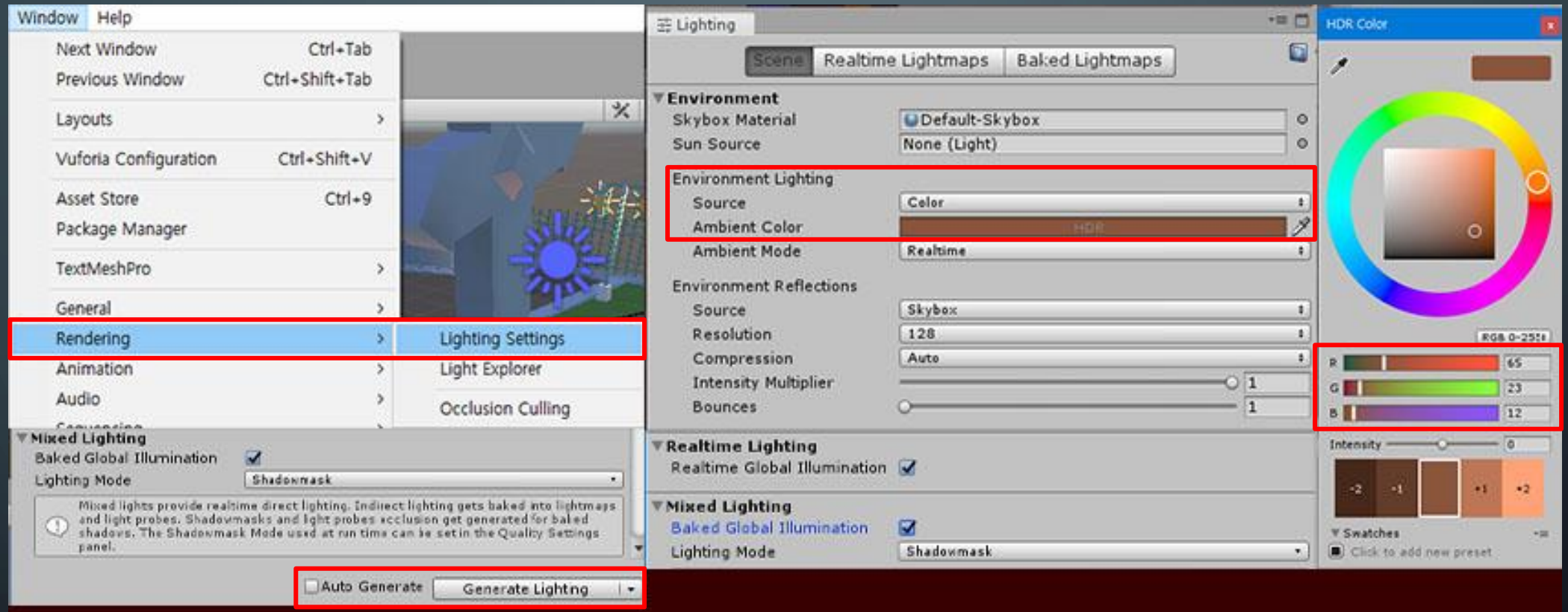
실제로 **Directional Light**를 삭제해서 실시간 광원이 없는 상태이다.

Light Object와 **Light Map**을 적용해 실제 빛이 비추는 강도와 효과를 통해 실시간 과원을 컨트롤하거나 배치하지 않고 적절한 효과를 줄 수 있다.



3. LIGHTING SETTING

AutoBaking이 적용되고 있는 셋팅을 조정해서 내가 원할 시점에 수동 **Baking**할 수 있게 옵션을 바꾸자.



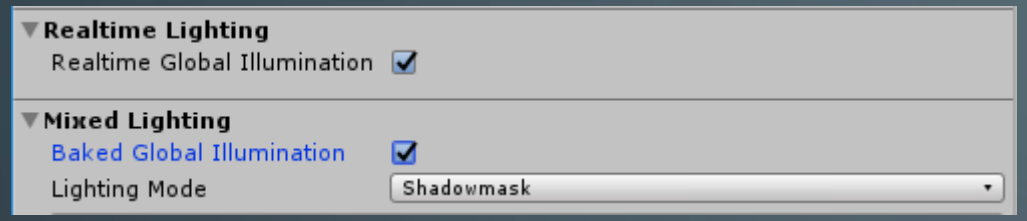
1. Window > Rendering > Lighting Settings 에서 Auto Baking을 적용하고 있는 Auto Generate 옵션체크를 해제하자.

2. 환경광을 적용하기 위해 Environment Lighting에서 Ambient Color를 Setting한다. HDR Color를 열어서 65, 23, 12로 설정해서 적용하자. Scene에 가장 기본으로 깔리는 빛.

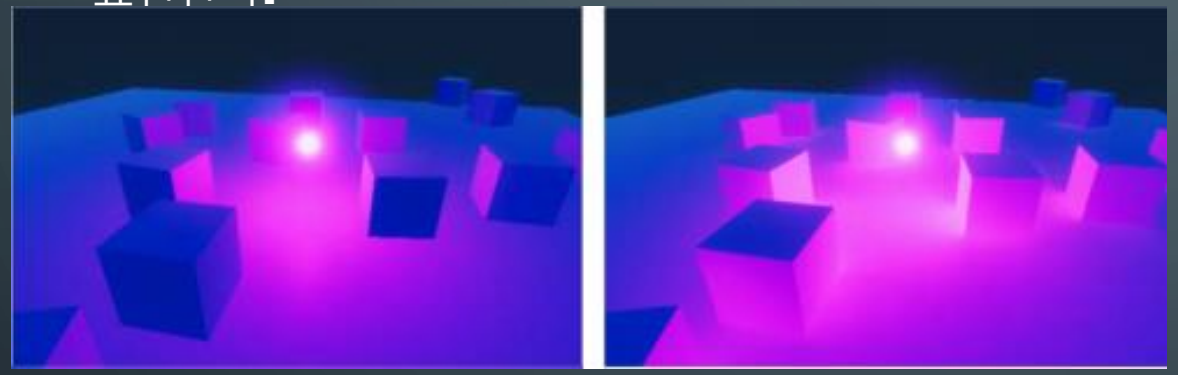
* 환경광 : 모든 GameObject에 적용되며, 모든 방향에서 같은 세기로 들어오기 때문에 그림자나 명암을 만들지 않는다, 환경광을 변경하면 게임의 전체 색 분위기를 변경할 수있다.

3. LIGHTING SETTING

글로벌 일루미네이션(Global Illumination)



- 물체의 표면에 직접 들어오는 빛 뿐만 아니라 다른 물체의 표면에서 반사되어 들어온 간접광까지 표현한다.



GI를 사용하지 않은 경우 **GI**를 사용한 경우

- **PC**성능으로도 실시간 글로벌 일루미네이션 옵션을 온전히 사용하기 힘들다.
  → 체크되어 있는 두 가지 옵션이 이미 적용이 되어 있고 여기서 리얼타임 글로벌 일루미네이션은 완전 실시간 글로벌 일루미네이션은 아니다.

Realtime Global Illumination, Bake Global Illumination이 체크되어 있는지 확인.

3. LIGHTING SETTING

실시간 글로벌 일루미네이션(Realtime Global Illumination)

- 빛의 세기와 방향 등이 달라졌을 때 그 변화를 간접광에 실시간으로 반영한다.
- **Light Map**을 여러 방향에 대해 생성한다.
  →여러 방향에 대한 빛의 예상 반사 방향과 예상 이동 결로 등의 정보를 미리 계산해서 저장한다.
  →미리 계산된 정보는 게임 도중 물체 표면에 들어오는 빛의 방향 등이 달라져도 간접광이 어떤 방향에 어떤 세기로 반사되어야 하는지 적은 비용으로 추측할 수 있으며, 광원의 변화를 실시간으로 간접광에 반영할 수 있다.
  →옵션을 설정할 경우 반드시 **Generate Lighting**을 통해 수동 **Baking**을 해줘야 한다.

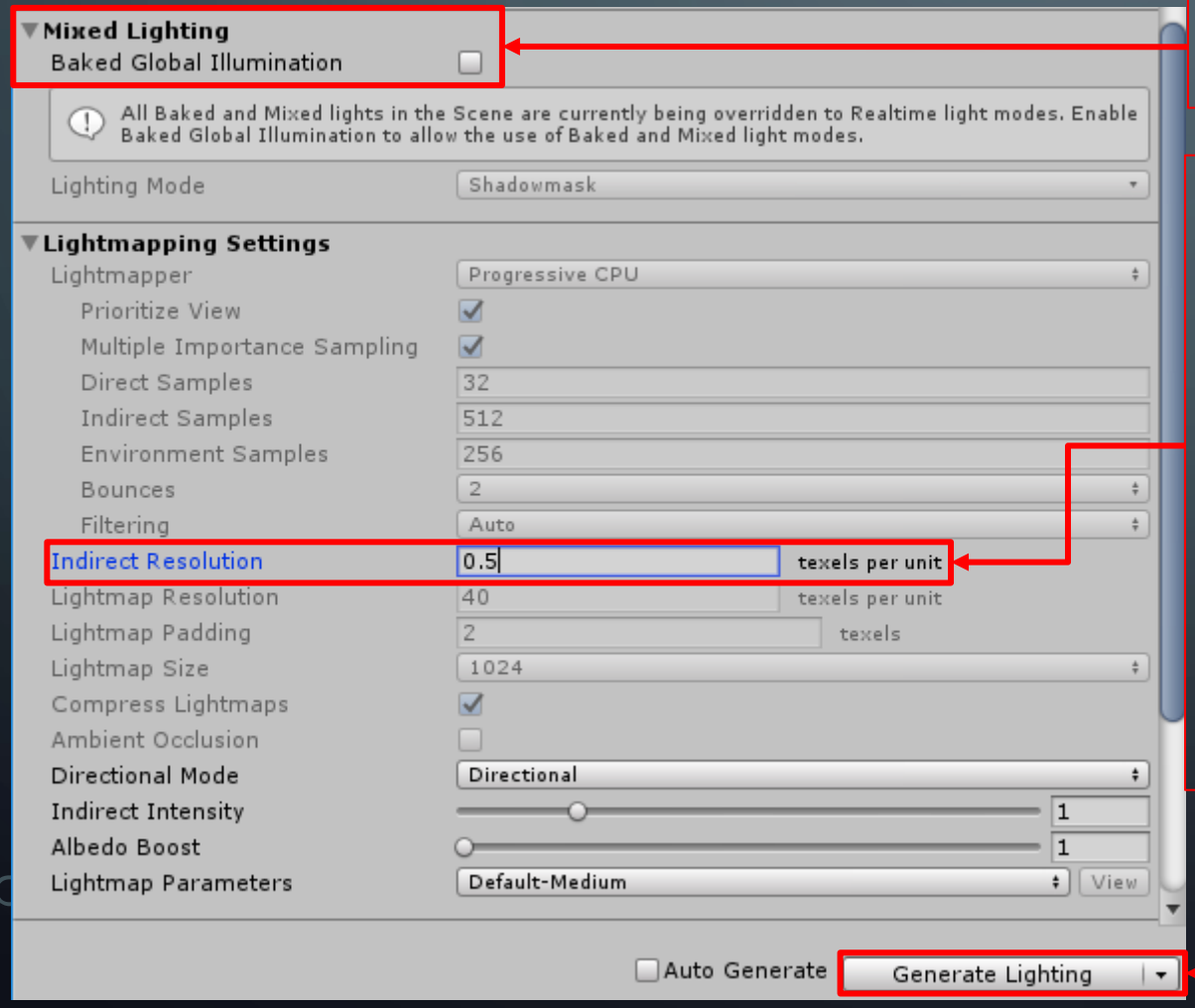
베이크된 글로벌 일루미네이션(Baked Global Illumination)

- 고정된 빛에 의한 간접광을 **Light Map**으로 **Bake**하여 **GameObject** 위에 미리 입힌다.
  →반영된 간접광 효과는 게임 도중에 실시간으로 변하지 않는다.
- 실시간 글로벌 일루미네이션보다 표현의 질과 런타임 성능이 더 좋다.
  →하지만 빛의 밝기나 방향이 게임 도중에 달라져도 간접광에 반영되지 않기 때문에 게임 도중에 갑자기 주변이 밝아지거나 어두워지면 이질감을 느낄 수도 있다.

3. LIGHTING SETTING

Global Illumination Option 설정하기

- 장점이 더 많은 **Baked Global Illumination**을 적용하는 것은 매우 오랜 시간 랜더링이 필요하고 빛의 변화에 따라 적용 받지 못하기 때문에 비용보다는 실시간 적용이 더 높은 가치를 가지고 있기 때문에 **Realtime Global Illumination**만 적용할 것이다.



Mixed Lighting의 **Baked Global Illumination** 체크 해제

Lightmapping Settings의 **Indirect Resolution**을 **0.5**로 변경

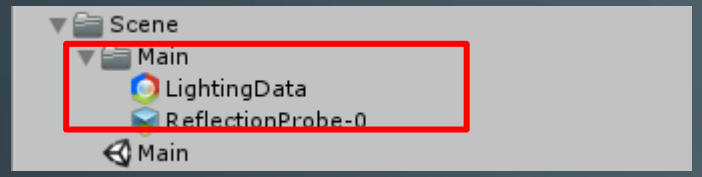
→ **LightMap**의 텍스쳐 해상도를 유닛당 **0.5**텍셀**(texel)**로 줄인 것이다**.** 이는 라이팅 효과의 정교함은 떨어지겠지만 로우 폴리 스타일의 **3D**모델을 사용하므로 그렇게 정교한 형태의 **Texture**가 필요하지 않기 때문

→ 텍셀**(texel) :** 텍스쳐의 화소를 말하며 화면의 **1**화소가 **1**픽셀이라면 텍스쳐의 **1**화소는 **1**텍셀이다**.**

Generate Lighting을 클릭

3. LIGHTING SETTING

- **LightMap Baking**이 완료되면 **Scene**폴더에 **Scene**이름**(**현재 **– Main)**으로 폴더가 생기고 그곳에 **Bake**된 **LightingData**와 **ReflectionProbe**가 저장된다**.**



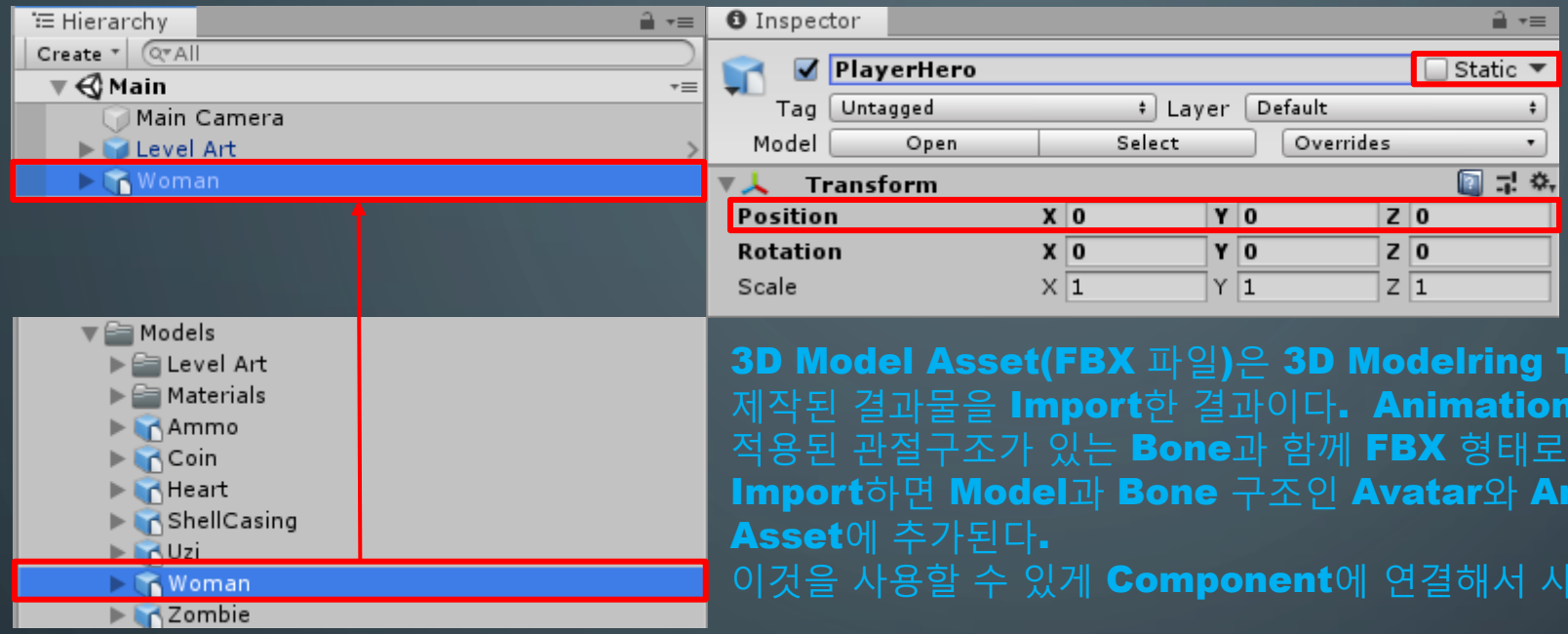
*** Global Illumination** 적용시 주의 사항

- **Static(**정적**) GameObject**에만 적용이 되기때문에 **Static** 체크박스를 체크해서 **Static**을 활성화 해야한다**.**
  → 정적으로 설정된 **GameObject**들은 게임 도중에 위치가 변경될 수 없다**.** 하지만 정적 게임 오브젝트에는 **Unity**가 상대적으로 다 많은 성능 최적화를 적용한다**.**
  → 유저에 의해 변화가 적용되는 상호작용 요소가 아니라면 **Static**으로 설정해서 최적화의 대상으로 적용 받으면서 **Ligth** 설정을 적용해보는 것도 좋다**.**

HUMANOID ANIMATION

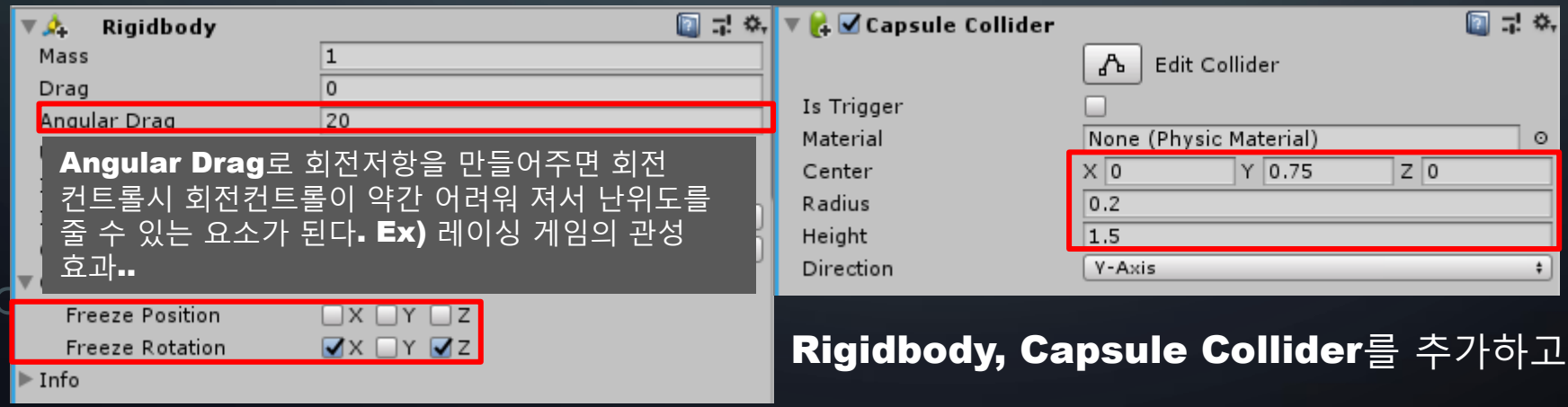
1. PLAYER CHARACTER추가

- **Medels > Woman**을 **Scene**에 추가하자.



3D Model Asset(FBX 파일)은 **3D Modelring Tool(ex 3D Max)**에 의해 제작된 결과물을 **Import**한 결과이다. **Animation**정보와 **Animation**이 적용된 관절구조가 있는 **Bone**과 함께 **FBX** 형태로 **Export**하고 이것을 **Import**하면 **Model**과 **Bone** 구조인 **Avatar**와 **Animation Clip**들이 **Asset**에 추가된다.
이것을 사용할 수 있게 **Component**에 연결해서 사용하면 된다.

- **FBX** 파일에 **Bone**이 없는 **Animation Clip**만 있을 수 있고 **FBX**가 아닌 **DS**파일 형태로 **Model**만 있을 수도 있다. 반드시 예제 형태처럼 구성된 것이 아니라는 것을 알고 있어야 한다.

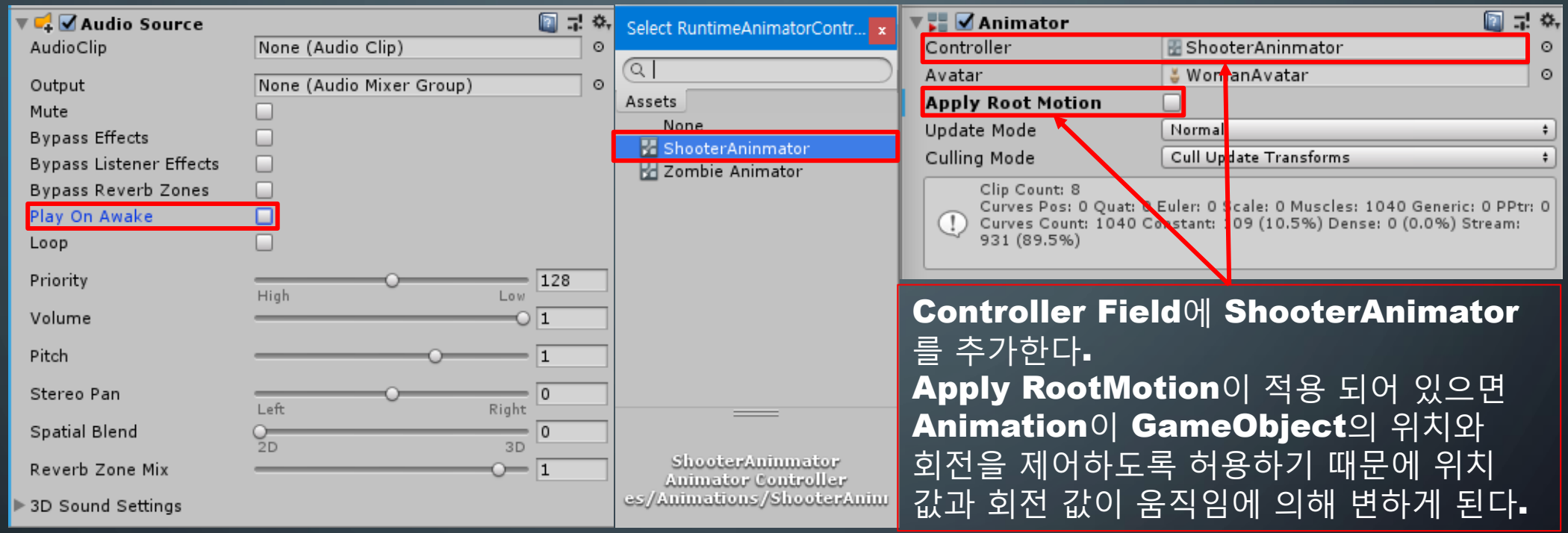


Angular Drag로 회전저항을 만들어주면 회전 컨트롤시 회전컨트롤이 약간 어려워 져서 난위도를 줄 수 있는 요소가 된다. **Ex)** 레이싱 게임의 관성 효과..

Rigidbody, Capsule Collider를 추가하고 설정하자.

1. PLAYER CHARACTER추가

- **Audio Source**를 추가하고 **Bone Animation**이 추가되어진 **Player Model**이기 때문에 **Animator**에 **Bone Avatar**를 적용하고 **Animator**를 적용하자.

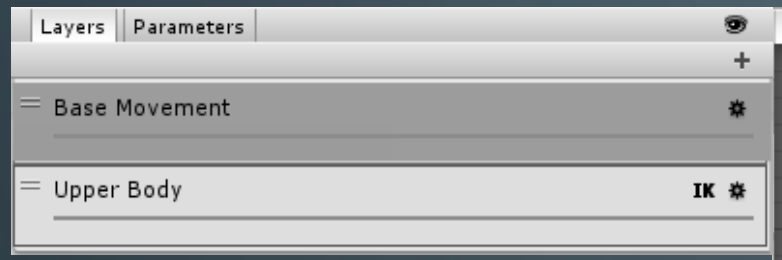


Controller Field에 **ShooterAnimator**를 추가한다.
Apply RootMotion이 적용 되어 있으면 **Animation**이 **GameObject**의 위치와 회전을 제어하도록 허용하기 때문에 위치 값과 회전 값이 움직임에 의해 변하게 된다.

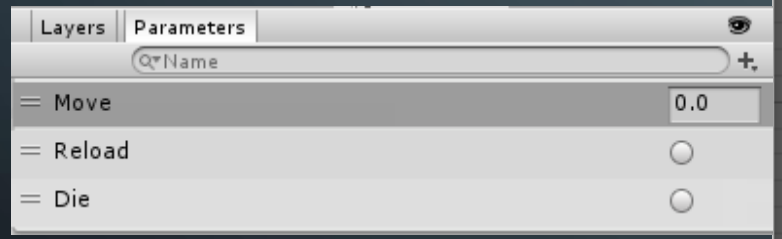
Animation이 앞으로 움직이는 모션이나 점프하는 동작에 의해 원래 위치보다 조금씩 벗어나서 변경되게 된다. 이런 현상을 방지하기 위해 실제로 애니메이션에 의해 적용되는 위치 값까지 모두 포함시키지 않기 위해 **Apply Root Motion**을 해제한다.

2. PLAYER CHARACTER ANIMATOR CONTROLLER

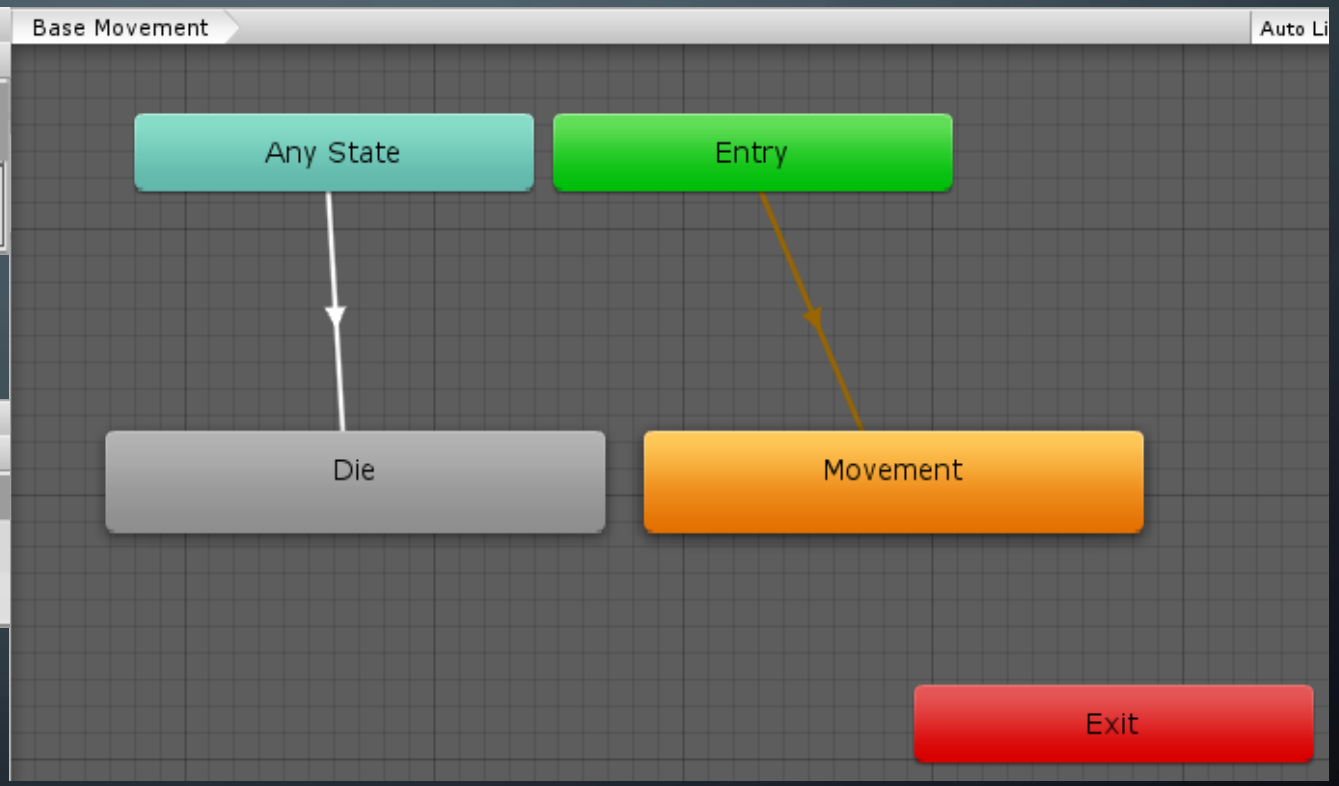
Animator View를 통해 미리 적용된 **Animator**를 확인해보자
→ 여러 개의 **Layer**로 **Animation**이 나뉘어져 있다.
→ 각 레이어가 **FSM**이 되는 것이고 이것을 병렬로 적용해 각각의 **FSM**상태를 동시에 적용하는 것이다.
→ **Ex)** 슈팅자세 **+** 뛰는 자세 **➔** 전체 자세는 뛰는 자세 상체 레이어를 슈팅자세로 변경하면 뛰면서 슈팅자세를 취할 수 있다.



AvatarMask라는 기법을 사용한 것이다. → 추후 설명.

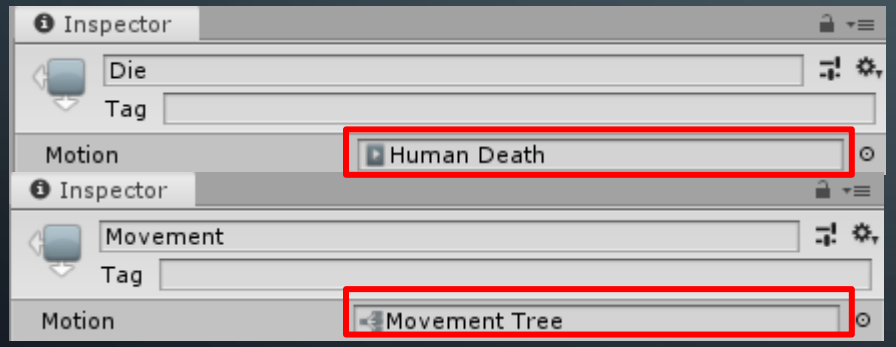


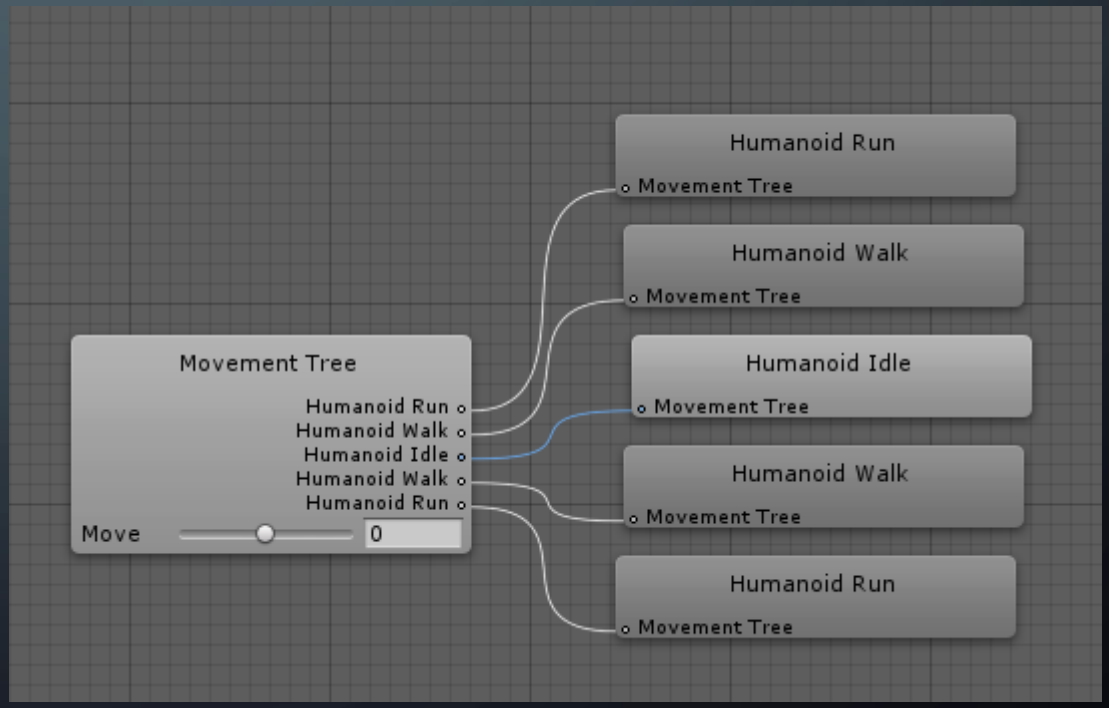
Move : 앞뒤 움직임에 관한 입력 값
Reload : 재장전을 알리는 트리거
Die : 사망을 알리는 트리거



3. BLEND TREE

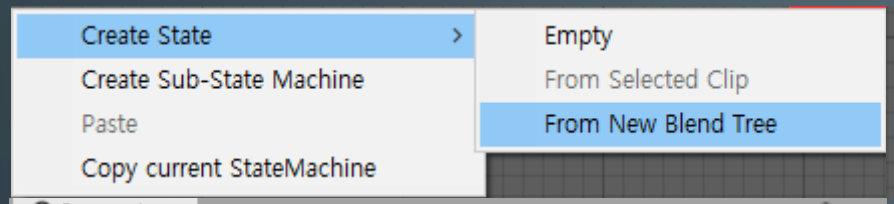
- 여러 상태를 미리 혼합해서 적용해둔 **Tree**를 가지고 상태 전환을 할 수 있다.
  →기본적으로 하나의 상태에 대해 하나의 **AnimationClip**을 적용해 나타나게 한다.
- 하나의 상태에 하나의 **Animation Clip**을 사용하면 한순간 상태가 너무 많아서 서로의 상태 전이 관계가 너무 복잡해질 수 있다. **Idle, Walk, Run**은 모두 **Movement**라고 하나의 상태로 체크하고 이 모두 **Move**의 값으로 체크할 수 있다. 서서히 각 상태를 섞어서 전환되어지는 상황이 될 수 있다.
- **Idle** 은 **Move** 값이 **0, Walk**는 **0.1~0.5, Run**은 **0.5 ~ 1**이라는 범위를 주고 있다고 가정할 때 **Walk**의 원래 **Animaiton**의 완전한 동작은 **0.5**의 순간이 될 것이다. **0.1~0.5** 사이에서는 완전한 걷기 동작으로 전환되는 동작들이 이어지고 있는 상환이 되는 것이다.
  → 마찬가지로 **1**이 전력질주의 **Animation**이다 그렇다면 **0.5**와 **1**의 상황에서는 서서히 전력질주의 동작으로 전환되는 상황의 동작이 반복되고 있는 것이다.

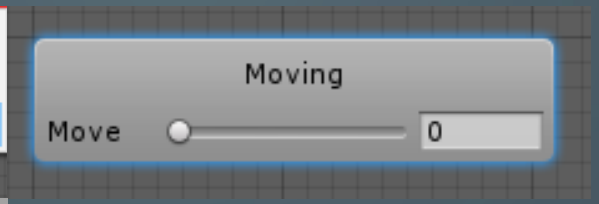


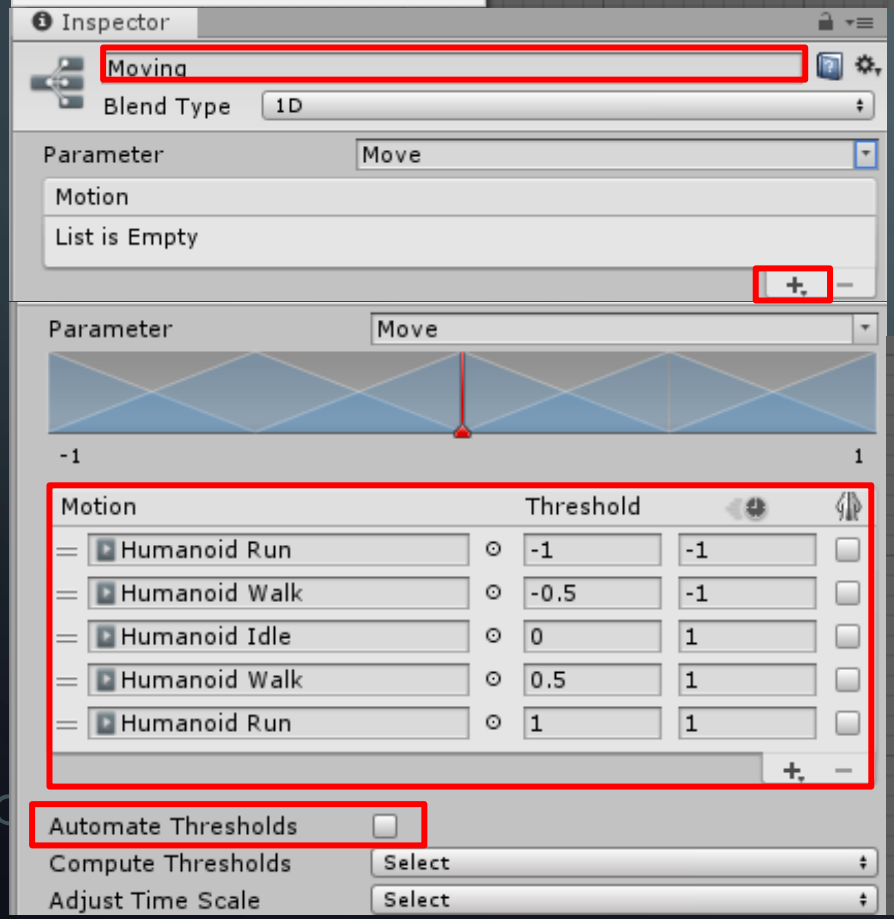


3. BLEND TREE

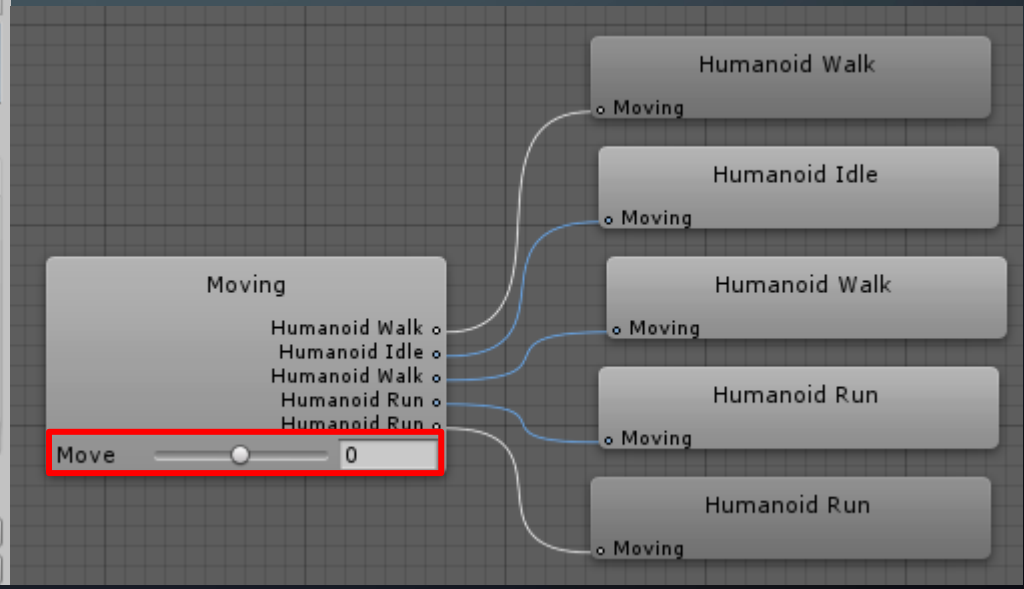
Movement를 더블 클릭하면 사용 할 **BlendTree**를 볼 수 있다.
BlendTree는 가중치가 필요하기 때문에 **float**이나 **int**형태의 **Parameter**만 적용가능하다.
Animator View > 빈 공간 우 클릭





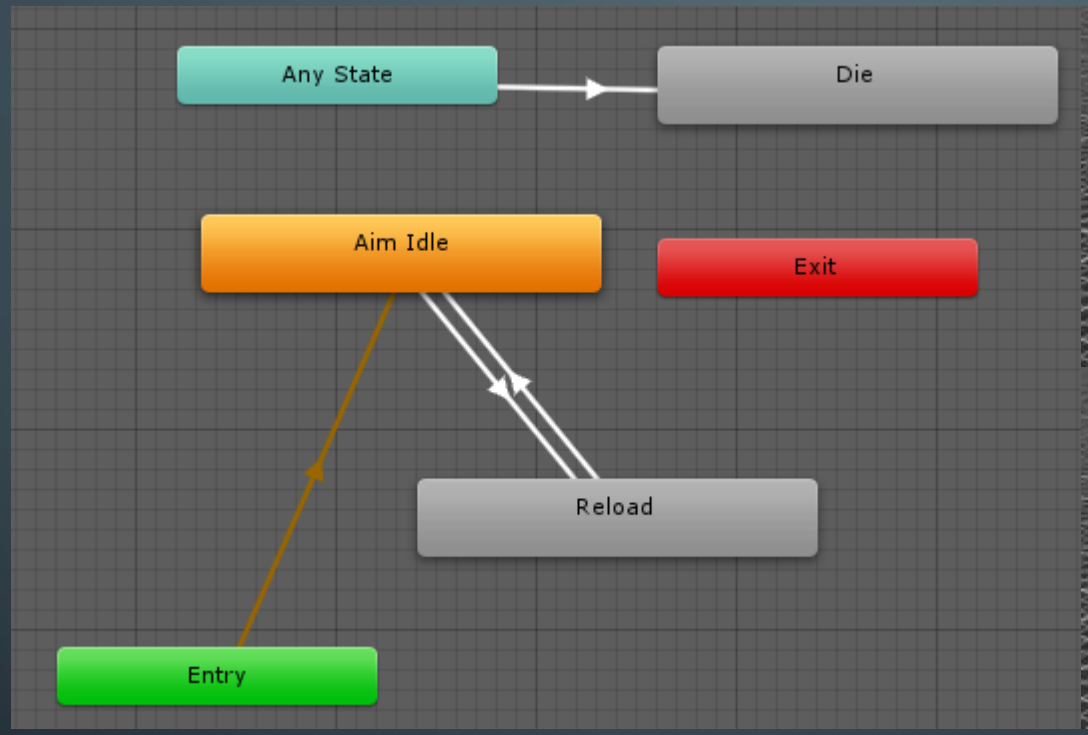


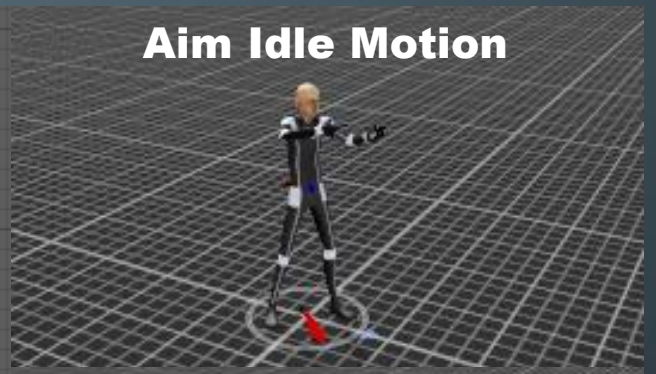
새로 생성한 **BlendTree**의 내부로 더블클릭을 이용해서 편집을 준비해보자.
이름을 설정하고 **Motion List**에서 **+**버튼을 눌러서 추가할 모션을 추가해보자.
AnimationClip을 추가하고 가중치 **Parameter**을 정하고 가중치를 설정해보자.



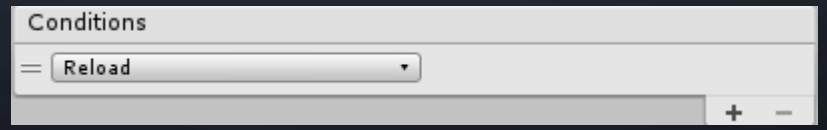
4. 병렬 레이어 – UPPER BODY LAYER

상체를 기본 **Body**의 움직임과 섞을 것이고 기본적으로 똑같이 적용되어야 하는 **Parameter**는 똑같이 표현해줘야 한다**.**



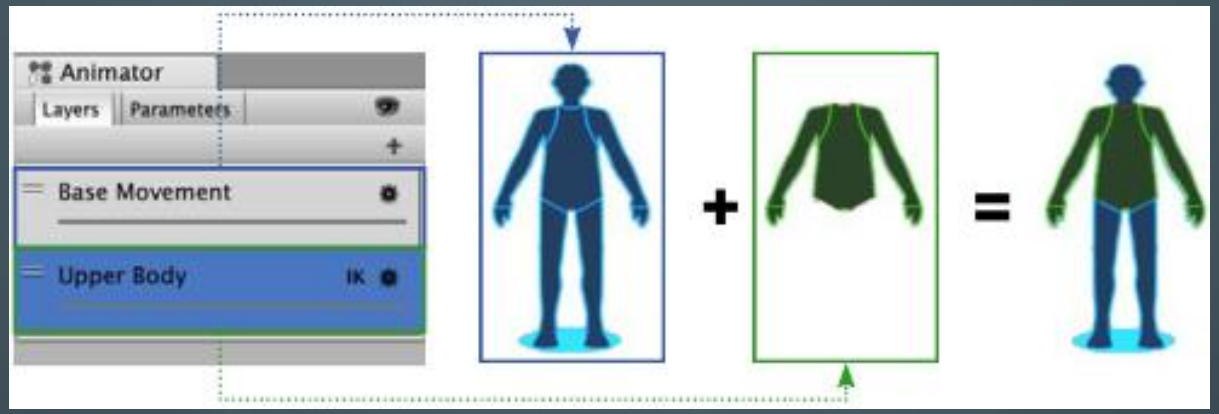


Aim Idle, Reload는 **Upper Body**에만 있다 이 상태일 때를 살펴보면 각각 **Reload Trigger**로 전환되고 **Base Movement**에서는 **Reload**가 **Trigger**가 없기 때문에 각각의 상태가 적용이 된다**.** 만약 같은 **Bone**이라면 상태가 전환되어 하나의 상태로만 적용되겠지만 서로 다른 **Bone** 상태인 것 처럼 **Avatar Mask**를 만들었기 때문에 가능하다**.**



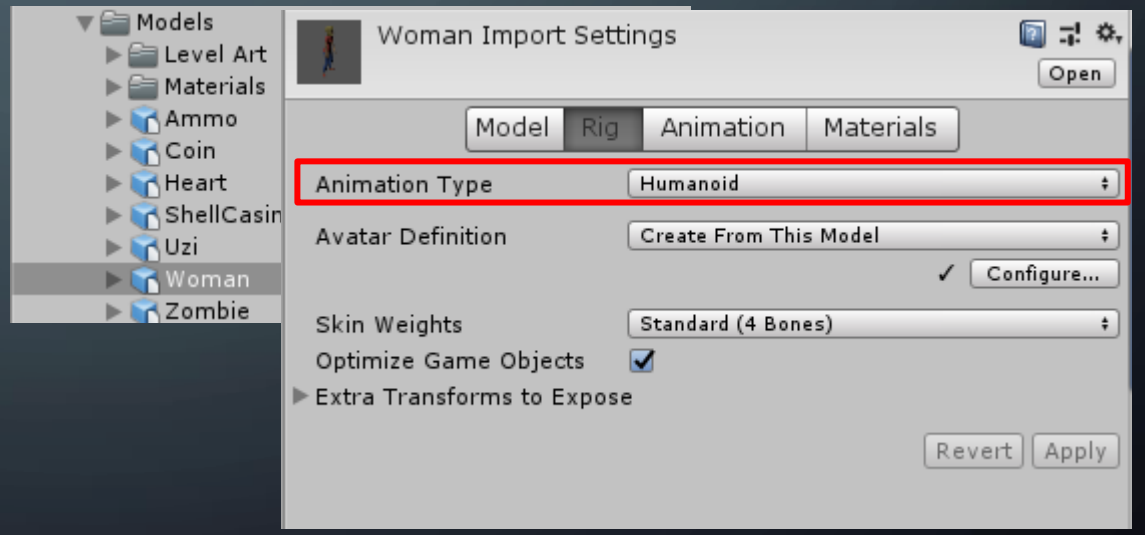
5. AVATAR MASK

Base Movement는 모든 **Bone Avatar**에게 적용되는 상태를...
Upper Body는 상체 **Bone Avatar**에만 적용되는 **Mask**를 적용...

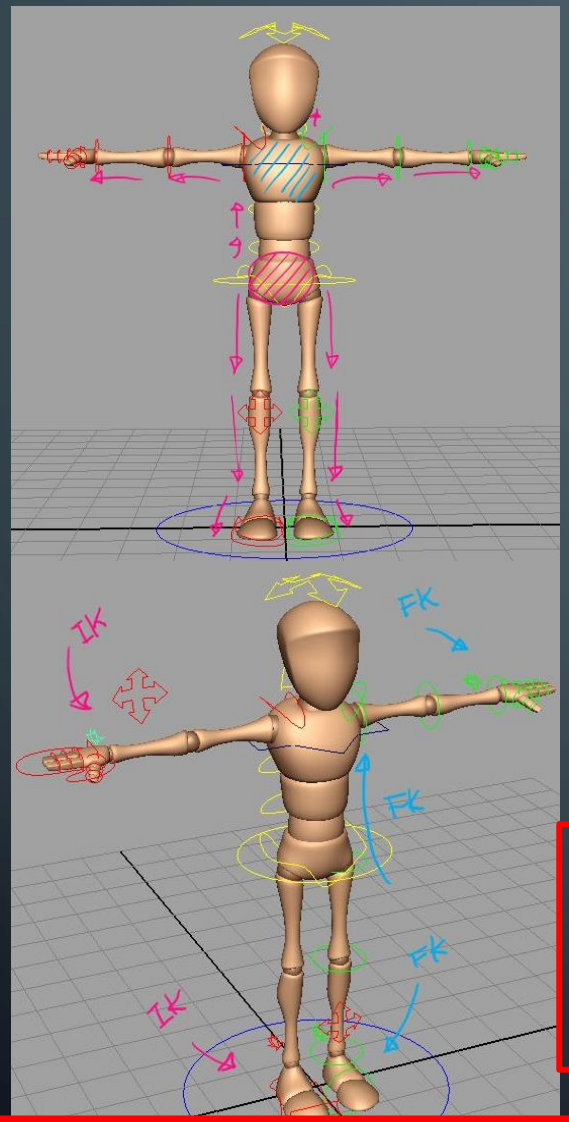


휴머노이드 닉(Humanoid Rig)

Bone 정보가 존재하는 **FBX Asset**일 경우 **Rig**정보를 **Humanoid** 설정이 가능하다.
주로 **2**족보행이 가능한 **Model**일 경우 **Bone**을 추가하여 **Animation**을 제작하게 되기 때문에 해당옵션에서 관절구조의 계층관계를 이용한 **Animation**이 설정과 제어가 가능하다.



5. AVATAR MASK



왼쪽은 **IK**적용 오른쪽은 **FK**적용

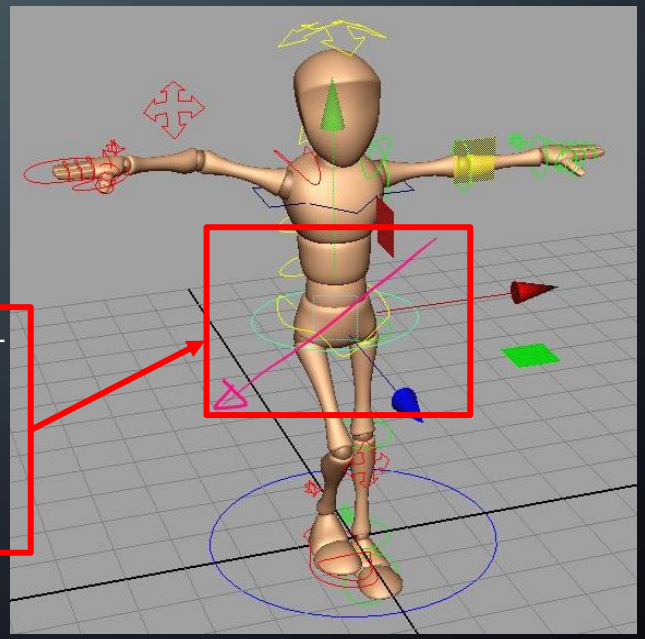
IK / FK

캐릭터를 움직이는 **Bone Animation**의 구조는 부모 – 자식 의 계층구조로 되어 있다**.** 실제 몸도 관절의 최상위 **Root**가 움직인다면 다른 신체의 하위 부위도 같이 이동이나 회전을 하게 된다**.**

FK(Forward Kinematics)
상위가 움직이면 하위가 따라 움직인다**.**
피규어의 관절구조**,** 사람의 신체구조
상반신이 아닌 모든 신체

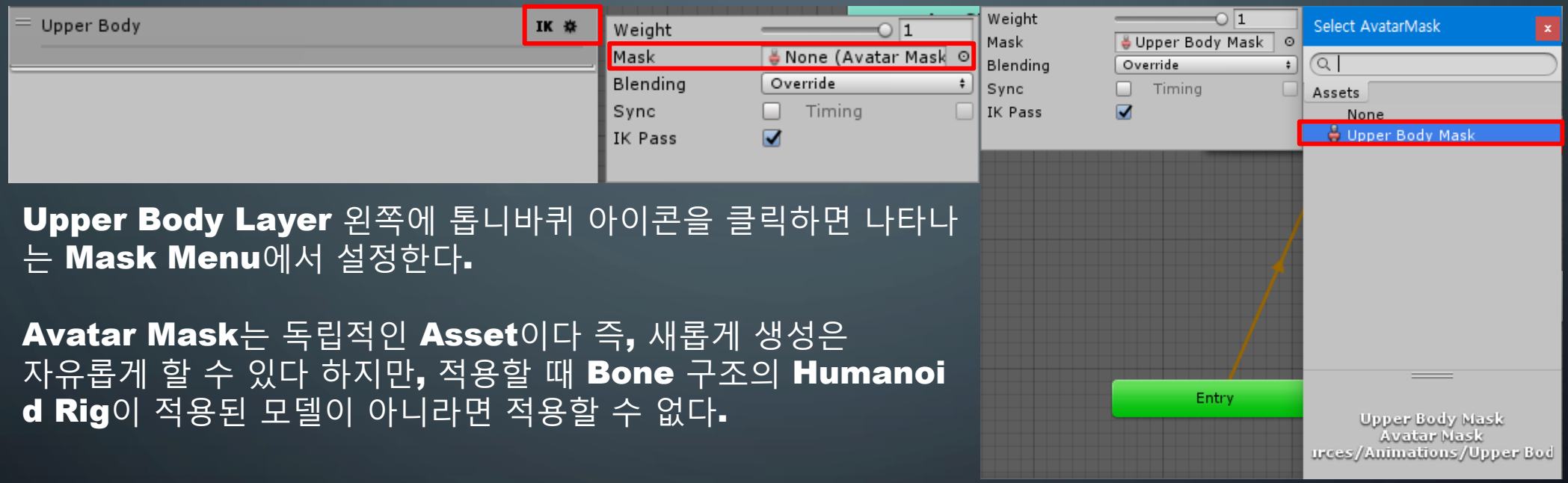
IK(Inverse Kinematics)
하위가 움직임이면 하위가 따라 움직인다**.**
마리오네트 인형**, Avatar..**
신체부위 중 발 – 지면의 기준이 되어야해서

최상위 부모**,** 골반을 화살표 방향으로 **(**왼쪽아래**)** 움직였다**.**
IK로 되어 있는 손목과 발바닥은 그대로 있고
오히려 반대로 그들의 부모인 팔과 다리가 움직여서 **IK**인 자식들의 위치를 유지 시킨다**.**



5. ANIMATION LAYER에 AVARTARMASK 적용

Upper Body에 **Avartar Mask**를 적용해야 한다.



Upper Body Layer 왼쪽에 톱니바퀴 아이콘을 클릭하면 나타나는 **Mask Menu**에서 설정한다.

Avatar Mask는 독립적인 **Asset**이다 즉, 새롭게 생성은 자유롭게 할 수 있다 하지만, 적용할 때 **Bone** 구조의 **Humanoid Rig**이 적용된 모델이 아니라면 적용할 수 없다.

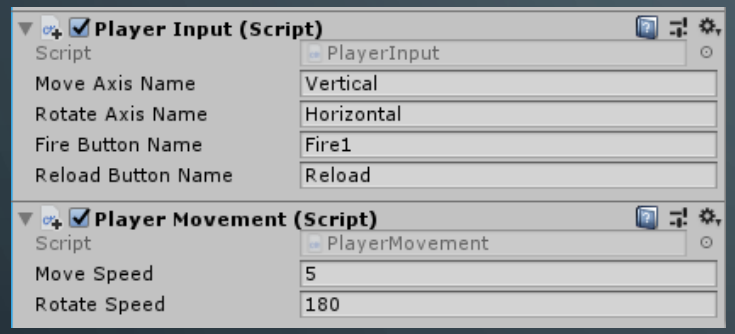
캐릭터 이동구현

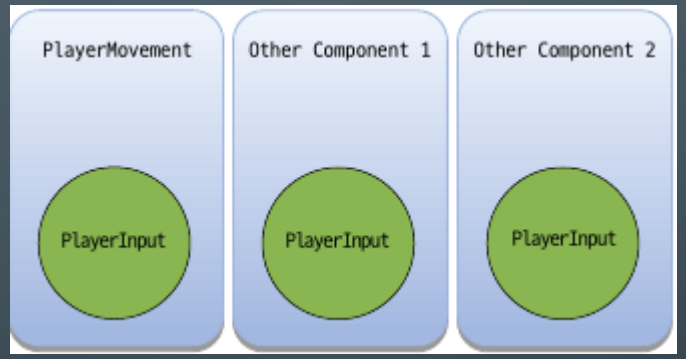
1. INPUT과 ACTOR 나누기

PlayerInput, PlayerMovement Script를 **PlayerCharacter**에 추가한다.

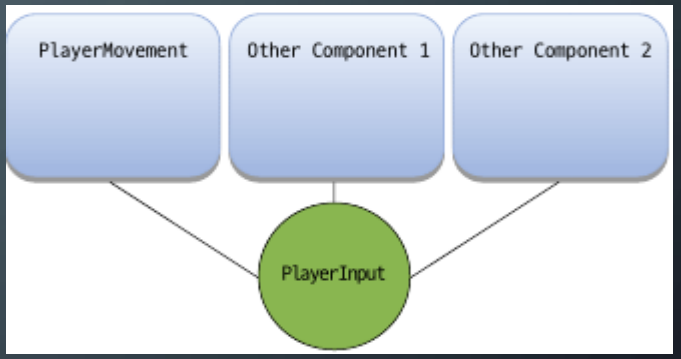
PlayerInput : Player의 입력을 체크한다 → **Controller**의 입력을 다른 **Component**에게 통지한다.
PlayerMovement : Player GameObject를 입력과 다른 상황을 통지 받아 **GameObject**를 컨트롤 한다.

두 개의 **Component**로 나누게 된 것은 확장성에 의한 코드 관리 때문이다.





Player Input을 제각각 처리하는 경우

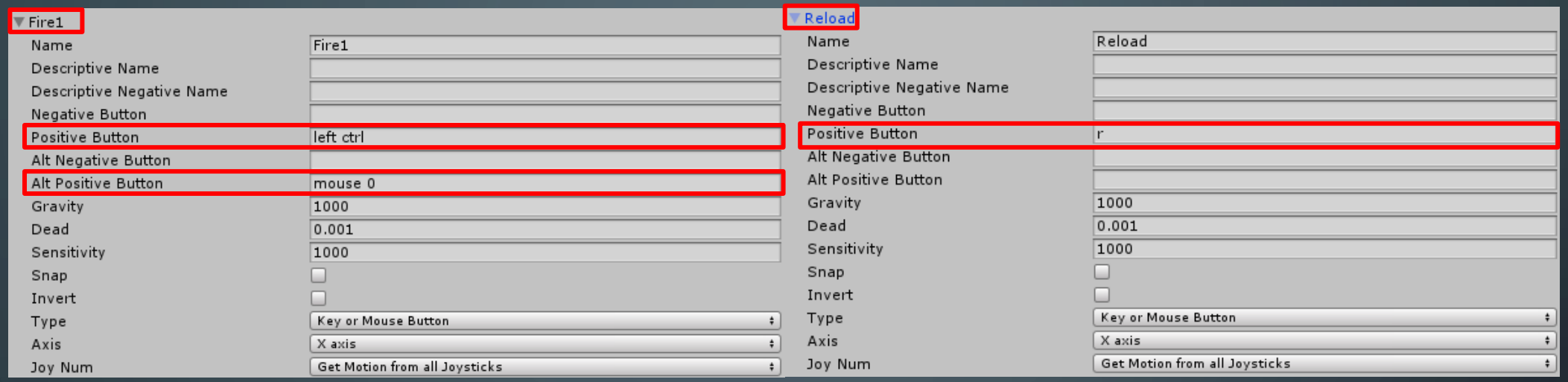


Player Input을 별개의 **Component**로 만드는 경우

게임을 제작할 때 플랫폼이 **PC**에서 모바일이나 콘솔로 바뀌거나 멀티플랫폼으로 변경되었을 때에 컨트롤 부분에 변경이 불가피하다 그렇다면 우리는 이를 해결 하기 위해 모든 **PlayerInput** 부분을 체크해야 한다.
하지만 우리는 이런 점을 개선하기 위해 객체지향의 **Class**를 사용하고 있고 **Component** 디자인이 적용된 엔진을 이용하고 있는 것이다.

1. INPUT CONTROLL OPTION 수정

Fire1과 **Reload**를 원하는 입력으로 수정하자.



입력을 **Code**에서 직접 할당하는 대신 미리 할당하여 준비해두자

```
public string moveAxisName = "Vertical"; // 앞뒤 움직임을 위한 입력축 이름
public string rotateAxisName = "Horizontal"; // 좌우 회전을 위한 입력축 이름
public string fireButtonName = "Fire1"; // 발사를 위한 입력 버튼 이름
public string reloadButtonName = "Reload"; // 재장전을 위한 입력 버튼 이름
```

움직임에 필요한 멤버들을 프로퍼티로 제공해서 사용 및 대입 가능하게 만들자.

```
public float move { get; private set; } // 감지된 움직임 입력값
public float rotate { get; private set; } // 감지된 회전 입력값
public bool fire { get; private set; } // 감지된 발사 입력값
public bool reload { get; private set; } // 감지된 재장전 입력값
```

Method형태로 제공되고 있기 때문에 private로 접근하여 값을 대입하기 위해 set Method를 제공하고 get 형태로 값을 받아 쓸 때, 변환 혹은 제어문을 통한 제한을 두어 필요한 상황에 맞춰 사용할 수 있다.

1. INPUT CONTROLL OPTION 수정

프로퍼티 예제

```
VolumeInfo info = new VolumeInfo();

info.bytes = 10000;
Debug.Log(info.kiloBytes);
Debug.Log(info.megaBytes);

info.megaBytes = 4;
Debug.Log(info.bytes);
```

```
private float m_bytes;
public float bytes
{
    get { return m_bytes; }

    set
    {
        if (value <= 0)
            m_bytes = 0;
        else
        {
            m_bytes = value;
        }
    }
}
```

```
public float megaBytes
{
    get { return m_bytes * 0.00001f; }

    set
    {
        if (value < 0)
        {
            m_bytes = 0;
        }
        else
        {
            m_bytes = value * 1000000f;
        }
    }
}
```

2. PLAYERMOVEMENT

사용 할 **Componet**의 참조

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

물리적인 정보의 갱신 주기가 기본적으로 **0.02**초에 맞춰 실행되기 때문에 **Update**보다 **FixedUpdate**를 이용하자**.**

```
private void FixedUpdate()
{
    //회전 실행
    Rotate();
    //움직임 실행
    Move();

    //입력값에 따라 애니메이터의 Move 파라이터값 변경
    playerAnimator.SetFloat("Move", playerInput.move);
}
```

Update 주기는 **Time.deltaTime**으로 체크가 가능하다**. FixedUpdate** 주기는 **Time.fixedDelta Time**으로 체크 가능하며 **Time.deltaTime**으로 체크하더라오 **fixedDeltaTime**으로 체크한 값이 전달된다**.**

2. PLAYERMOVEMENT

정면으로 상대적 이동 방향 값 = 방향 * 속력 * 시간, **playerInput.move**의 값으로 전진 후진 결정.

```
// 입력값에 따라 캐릭터를 앞뒤로 움직임
private void Move()
{
    //상대적으로 이동할 거리 계산
    Vector3 moveDistance =
        playerInput.move * transform.forward * moveSpeed * Time.deltaTime;

    //리지드바디를 이용해 GameObject 위치 변경
    playerRigidbody.MovePosition(playerRigidbody.position + moveDistance);
}
```

회전 값 = 속력 * 시간, **playerInput.rotate**의 값으로 시계, 반시계 방향 회전 결정

```
// 입력값에 따라 캐릭터를 좌우로 회전
private void Rotate()
{
    //상대적으로 회전할 수치 계산
    float turn = playerInput.rotate * rotateSpeed * Time.deltaTime;

    //리지드바디를 이용해 GameObject 회전 변경
    playerRigidbody.rotation =
        playerRigidbody.rotation * Quaternion.Euler(0, turn, 0f);
}
```

회전은 곱이 회전 값 추가이다 **Matrix * Matrix**

CINEMACHINE을 이용한 FOLLOW CAM구현

1. CINEMACHINE

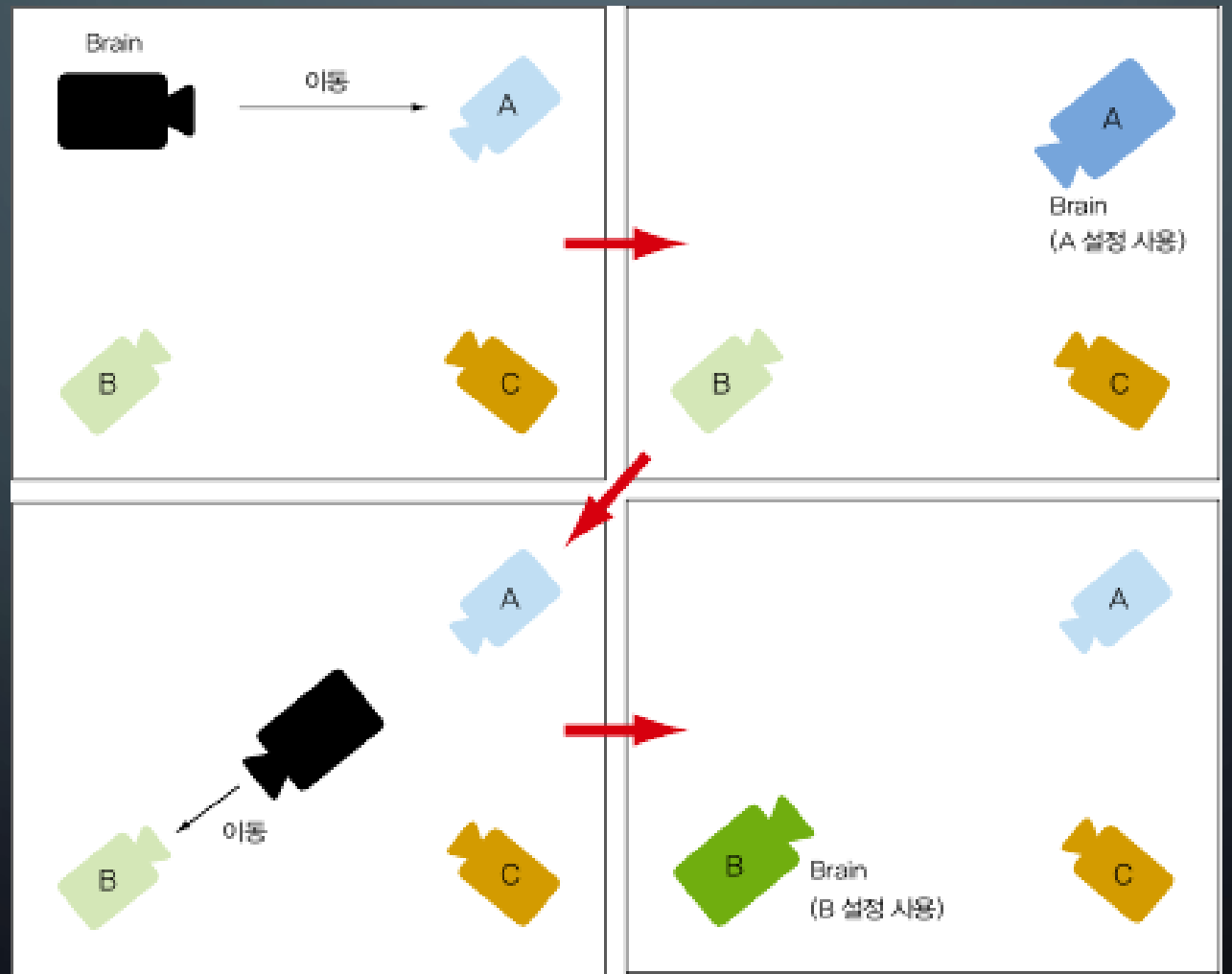
- 카메라의 움직임을 손쉽게 제어하는 **Unity Standard Package**
- 카메라 연출에 필요한 코드와 조정 작업 대부분을 대체할 수 있다.
- 레이싱, 어드벤처, **TPS**등 장르마다 고유한 카메라 동작을 별다른 **Script** 작성 없이 구현할 수 있다.
  →카메라 조작에 대한 부담을 해결하는 도구일 뿐이고 실제적으로 어떤 원리와 방법으로 구현이 되는지를 파악해 나가면서 직접적으로 본인이 카메라를 코드로 컨트롤 할 수 있게 되어야 한다.
- **Cinemachine**의 **Component**들은 카메라 초점, 화면상의 피사체 배치, 추적의 지연시간이나 카메라 흔들림, 여러 카메라 사이에서의 전환 등 카메라 연출과 관련된 다양한 수치를 제공한다.
  →연출 의도에 맞춰 변경하고 카메라가 추적할 대상만 지정하면 **Cinemachine Camera**가 알아서 목표물을 화면에 담아낸다.

Cinemachine의 원리

- **Cinemachine**에서는 크게 두 종류의 카메라로 제공되어 진다.
  →**Brain Camera, Virtual Camera**
- 여러 대의 **Camera**를 준비해서 다각도의 **Camera**로 연출하고 있다가 지금 현재 **Main Camera**에 보여질 **Camera**가 어떤 것인지 설정하면서 다각도의 변환을 하는 원리를 이용하는 것이다.
  →방송국에서 **Studio**에 여러 대의 카메라로 찍고 있다가 조정실에서 카메라 전환 버튼으로 시청자들에게 보여줄 **Camera**를 전환해가면서 보여주는 기법이랑 동일하다고 보면 된다.

1. CINEMACHINE

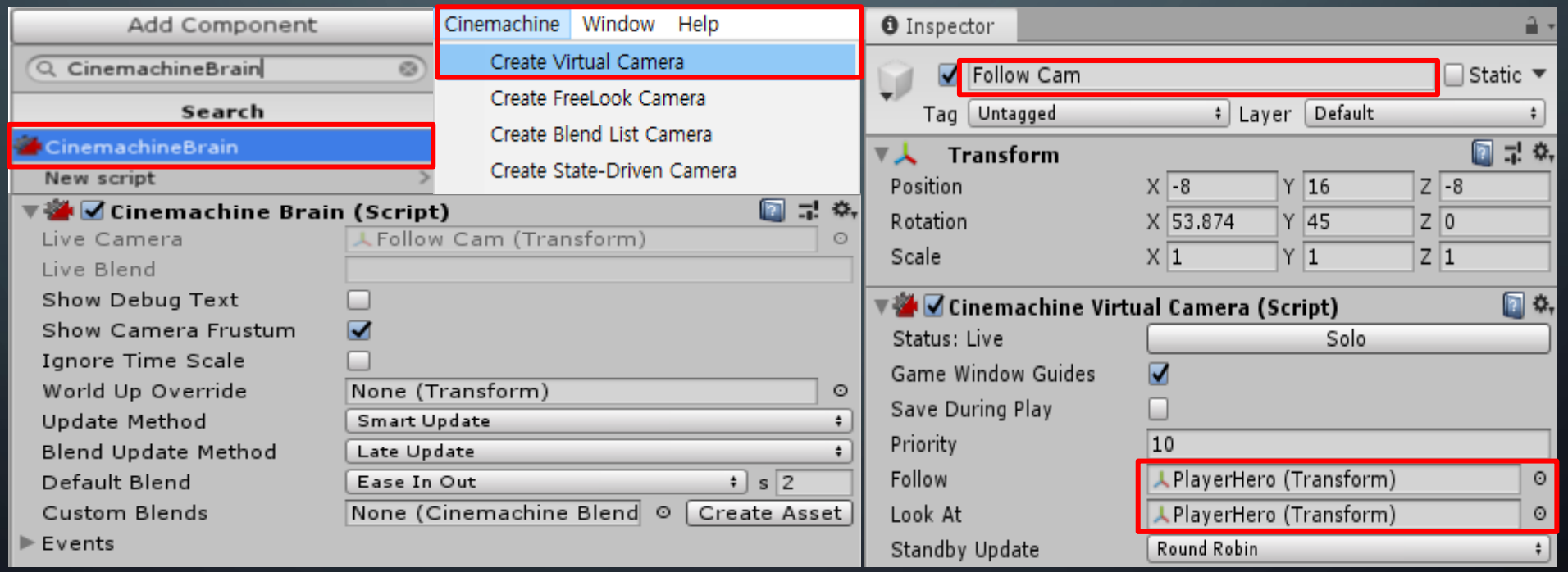
- **Scene**에 **Virtual Camera A, B, C**를 배치 했다고 가정해보자.
- **Brain Camera**는 한 번에 하나의 **Virtual Camera**만 현재 활성화된 **Camera**로 사용 가능하다.
- **Brain Camera**가 **Virtual Camera A**를 현재 카메라로 활성화하여 사용했다고 가정할 때, **Brain Cam era**는 **Virtual Camera A**의 위치로 이동하고, **Virtual Camera A**의 모든 설정을 자신의 설정으로 사 용한다.



2. CINEMACHINE SETTING

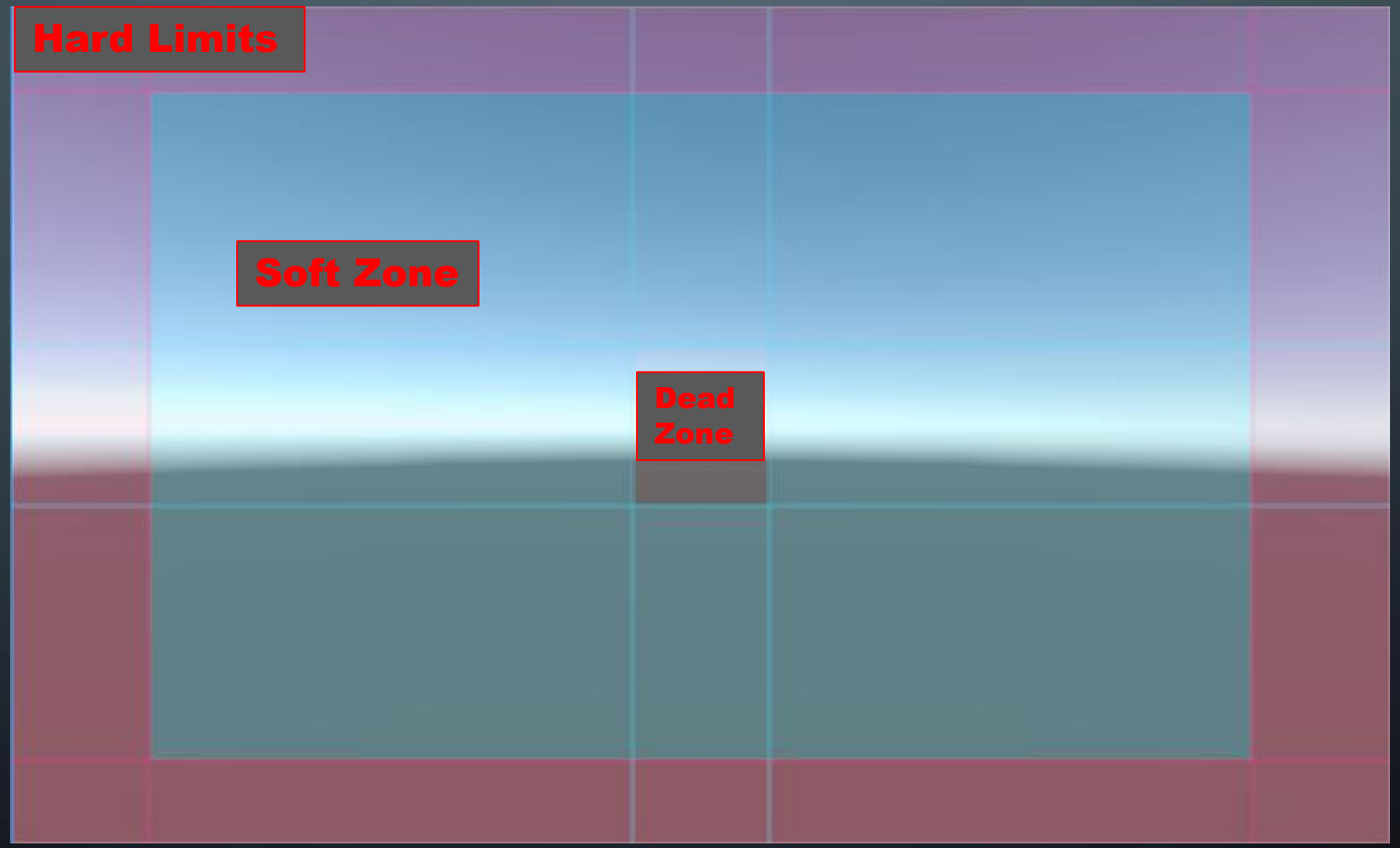
Brain Camera와 **Virtual Camera** 만들기

1. 하이어라키 창에서 **MainCamera GameObject**에 **Add Component**로 **Cinemachine Brain** 추가
2. 새로운 **Virtual Camera**를 **Scene** 배치(**Menu Bar > Cinemachine > Create Virtual Camera**)
3. **Follow Cam**으로 설정하고 **Follow Field**와 **LookAt Field**에 따라다닐 대상을 할당해 준다.



2. DEAD ZONE, SOFT ZONE, HARD LIMITS

- **Target**을 할당하면 **Camera**의 자연스러운 추적을 구현하는 데 사용하는 **Soft Zone, Hard Limits**과 **DeadZone** 영역이 표시된다.
  → **Virtual Camera Component**의 속성값을 변경하거나 **Game View**의 **Line**들을 드래그 해서 조정할 수 있다.



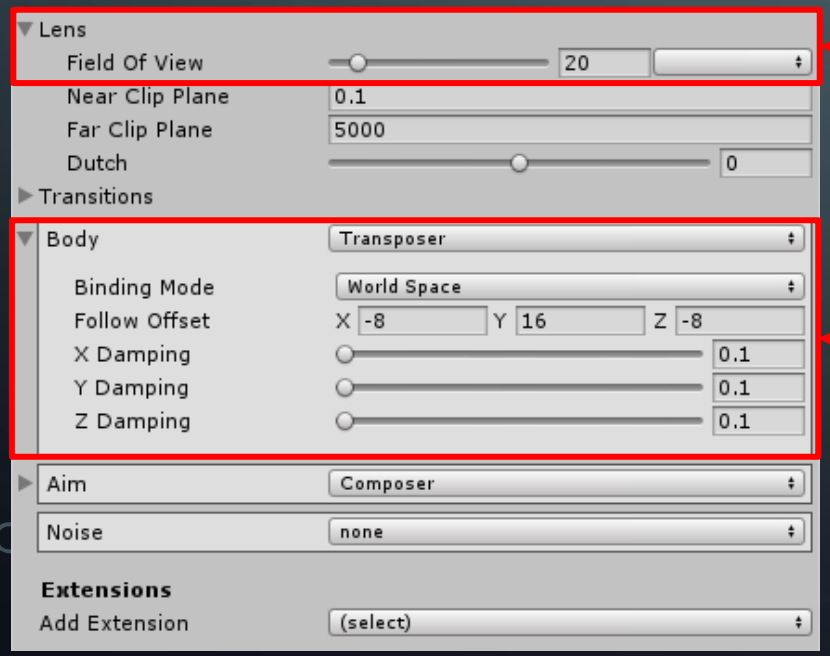
2. DEAD ZONE, SOFT ZONE, HARD LIMITS

- 영역별로 지정한 것은 추적하고 있는 **Camera**들이 회전하거나 변화할 때 자연스럽게 추적 할 수 있는 정도를 설정하는 역할을 한다.

- 주시하는 물체가 **Dead Zone** 영역에 존재하는 동안 카메라는 회전을 하지 않다가 화면의 **Soft Zone**에 있다면 물체가 화면의 조준점**(Aim)**에 오도록 카메라가 부드럽게 회전한다.
- 만약**,** 물체가 너무 빠르게 움직여 화면의 소프트 존을 벗어나 **Hard Limits**에 도달하려 한다면 카메라는 빠르게 회전해서 **Soft Zone**을 벗어나지 않게하고 **Dead Zone**으로 진입하면 회전을 멈춘다.

Virtual Camera의 **Body**와 **Aim** 설정

1. **Field Of View**를 **20**으로 변경하여 카메라의 시야각을 설정한다.
2. **Body > Binding Mode**를 **World Space**로 변경하고 **Follow Offset**을 **(-8, 16, -8)**로 변경한다.
3. **X Damping, Y Damping, Z Damping**을 **0.1**로 변경한다.

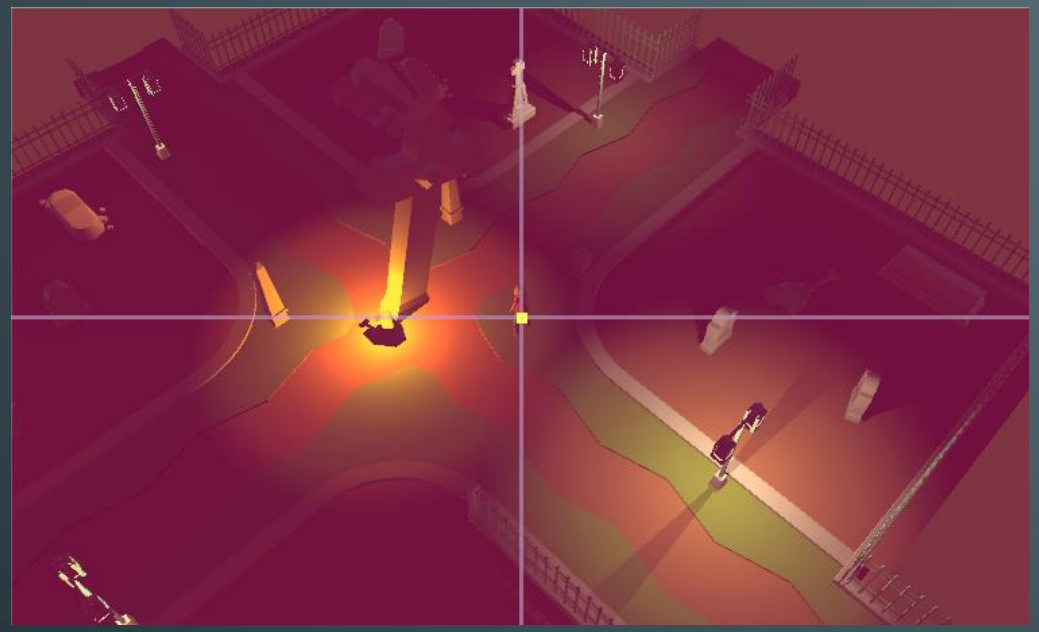


Field Of View를 **20**으로 변경

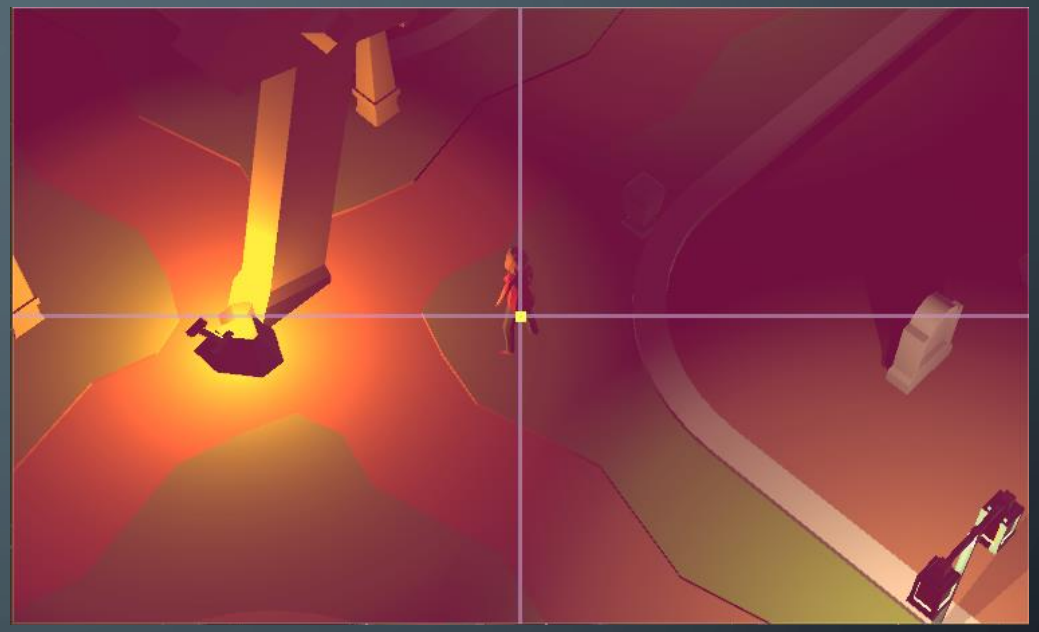
Binding Mode를 **Word Space**로 변경
Follow Offset (-8, 16, -8)
X Damping : 0.1
Y Damping : 0.1
Z Damping : 0.1

2. DEAD ZONE, SOFT ZONE, HARD LIMITS

Field Of View(FOV) : 시야각을 나타내는 것으로 값의 변화에 따라 카메라에 들어오는 모습이 달라진다.



Fov : 40



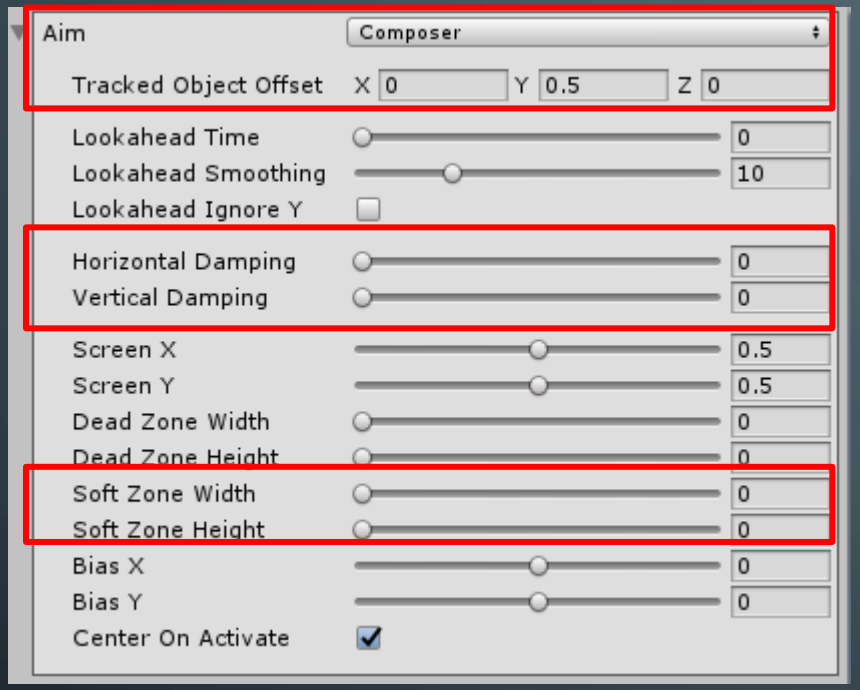
Fov : 20

Body Parametar는 **Follow** 대상을 어떻게 따라 갈 것인지 결정하는 요소 이다.

Binding Mode : 전역공간에 존재할지 대상의 로컬 공간에 존재할지 설정한다.
Follow Offset : 대상으로 부터 얼마나 떨어져 존재하는지 설정한다.
?? Damping : 추적해서 따라갈 때 얼마의 크기로 저항을 줄 것인지 설정한다.
→ 값이 적을수록 빠르게 따라간다.

2. DEAD ZONE, SOFT ZONE, HARD LIMITS

1. **Aim**의 **Traked Object Offset (0, 0.5, 0)**으로 변경
2. **Horizontal Damping**과 **Vertical Damping**을 **0**으로 변경, **Soft Zone, Width**와 **Soft Zone Height**를 **0**으로 변경



Tracked Object Offset Field는 원래 추적 대상에서 얼마나 더 떨어진 곳을 조준할지 결정한다.

회전에 대한 제어 값을 **0**으로 해서 바로 따라 갈 수 있게 하였다.

Soft Zone또한 제거해서 지연시간을 가지고 변화하지 않게 했다.