



UNITY -CHAPTER4-

SOUL SEEK

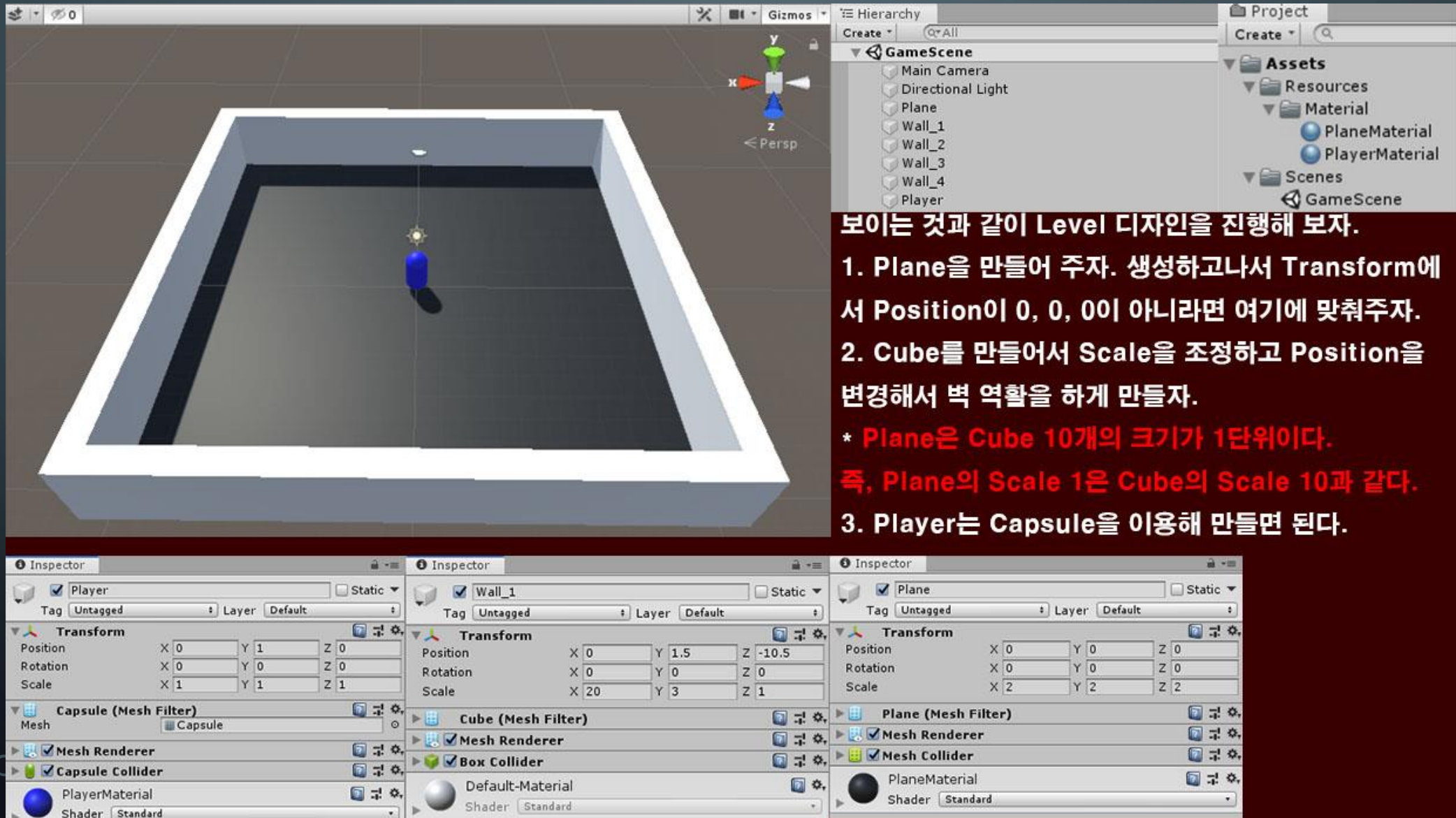
목차

1. 월드구성(Level Design)
2. Camera 설정 & Player 제작
3. Player 입력감지

월드 구성 (LEVEL DESIGN)

1. 월드 구성(LEVEL DESIGN)

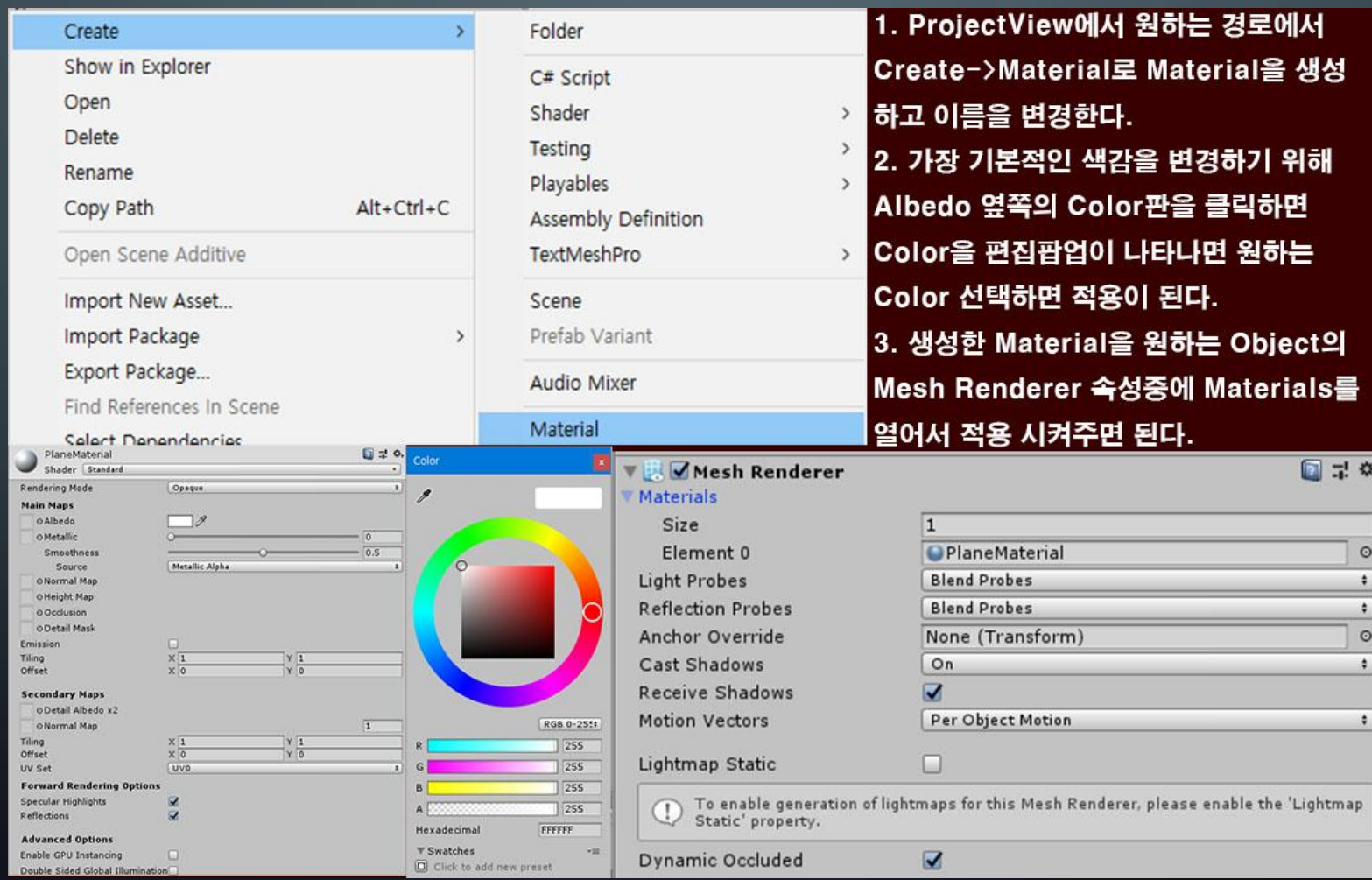
기본적인 월드 구성을 알아보자.



1. 월드 구성(LEVEL DESIGN)

GameObject에 Color를 적용시켜보자.

- **3D World**에서 **Object**가 보이기 위해서는 **Material**이 필요하다.
- **Unity**에서 제공하고 있는 **Default Object**은 **Default Material**이나 **Default Texture**을 적용 받고 있다.



The screenshot displays the Unity interface with three panels visible:

- Hierarchy Panel:** Shows the 'Create' menu open, with 'Material' selected at the bottom.
- Inspector Panel:** Shows the 'Material' properties for 'PlaneMaterial'. The 'Color' property is highlighted, and a color picker is open, showing a red color.
- Inspector Panel:** Shows the 'Mesh Renderer' component. The 'Materials' list contains 'PlaneMaterial'.

1. ProjectView에서 원하는 경로에서 Create->Material로 Material을 생성하고 이름을 변경한다.

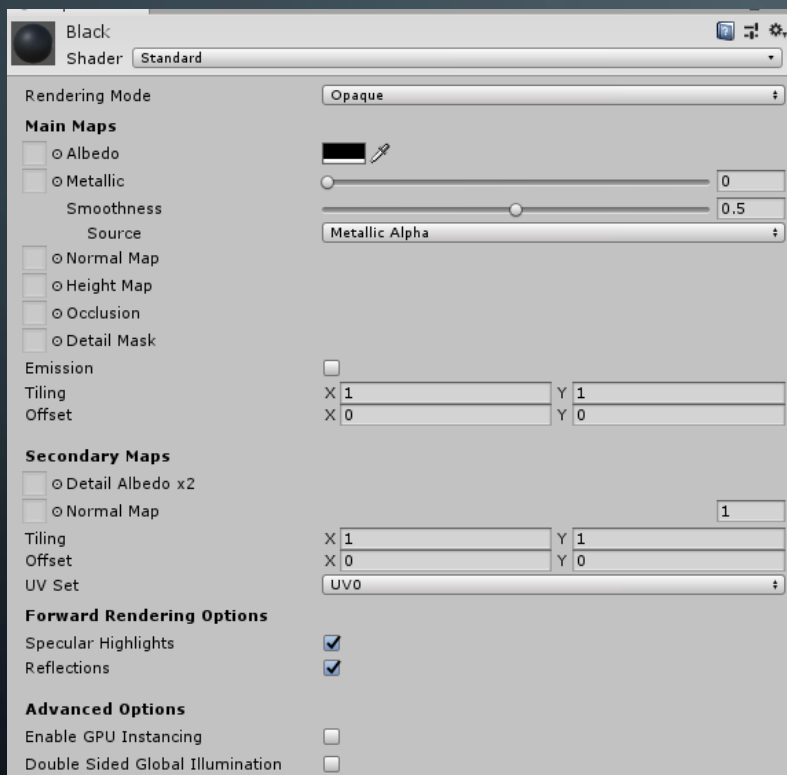
2. 가장 기본적인 색감을 변경하기 위해 Albedo 옆쪽의 Color판을 클릭하면 Color를 편집팝업이 나타나면 원하는 Color 선택하면 적용이 된다.

3. 생성한 Material을 원하는 Object의 Mesh Renderer 속성중에 Materials를 열어서 적용 시켜주면 된다.

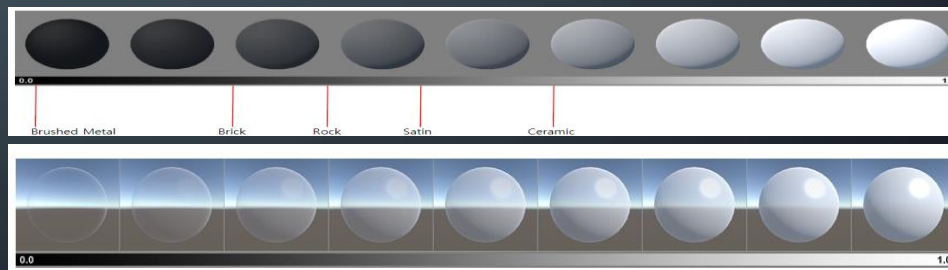
1. 월드 구성(LEVEL DESIGN)

Material

- 물체의 표면을 정의하는 **Component**이다.
- **Texture**와 **Light, Shader**로 표현해서 다양한 표면 효과를 나타낸다.
- 여러가지 속성을 제공받고 설정 할 수 있는 공간(**Shader**의 종류에 따라 제공받는 속성이 다름)



- **Rendering Mode** : 투명도를 사용할지 여부와 사용할 경우 사용 할 **Blending** 모드 유형을 선택 할 수 있다.
 - **Opaque** – 기본 속성이며 투명하지 않은 오브젝트에 적합하다.
 - **Cutout** – 구멍이나 너덜너덜한 잎, 천같은 투명도를 가져야 할 때
 - **Transparent** – 투명한 플라스틱이나 유리 등의 현실적인 투명도, 텍스처와 **Tint Color** 알파 값에 근거한 알파 값을 가짐
 - **Fade** – 투명 값이 오브젝트를 완전히 **Fade Out**할 수 있는데 해줍니다 **Specular** 하이라이트나 반사에 포함되고 **Fade Out**이나 인 효과에 유용하다. 대신 현실적인 투명도를 가진 사물엔 부적합 하다.
- **Albedo** : 표면의 기본 색상을 제어하고 알파 값은 투명도를 제어한다.



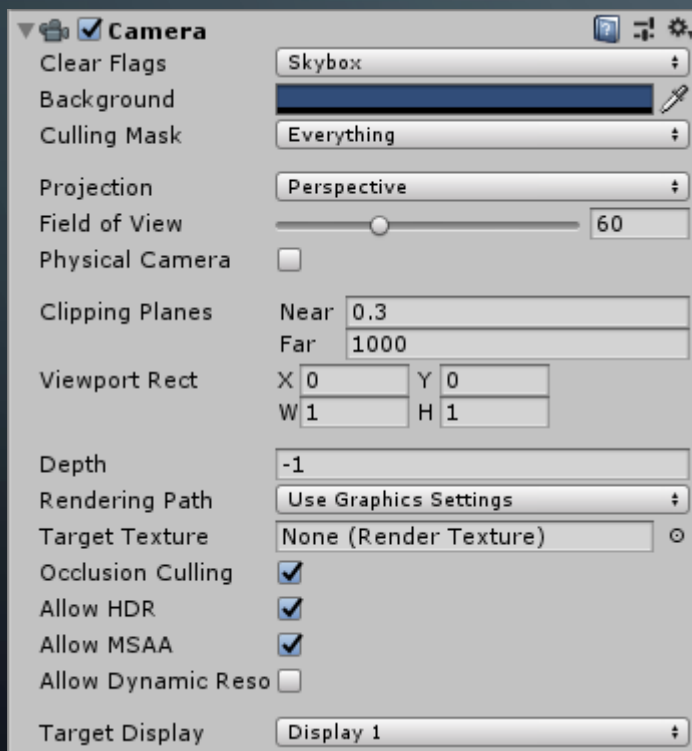
- **Specular & Metallic** : **Specular** – 하이라이트 광택과 **Tint Color**를 통해 재질의 광원과 반사를 설정한다. **Metallic** – 금속성을 얼마나 가지는가를 설정한다. 두가지는 같이 공유되는 프로퍼티가 아니다.

CAMERA 설정과 PLAYER제작

2. CAMERA 설정과 PLAYER제작

Camera

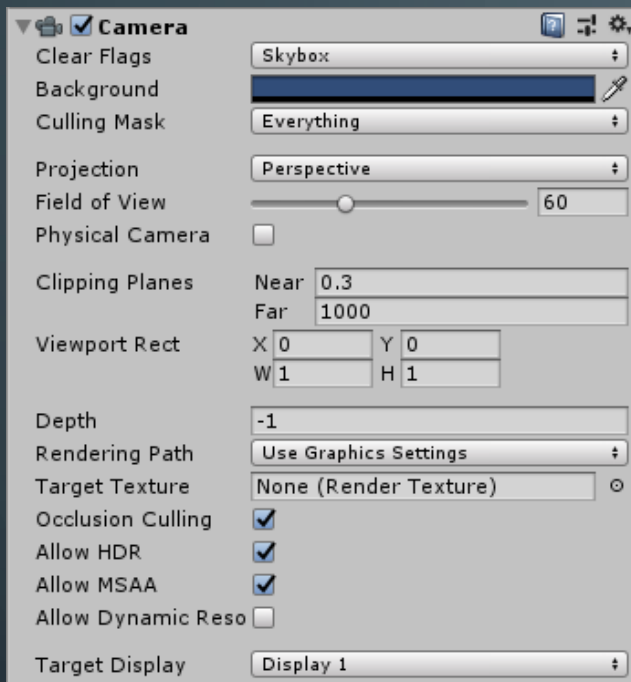
- 월드를 캡처해서 플레이어에게 보여주는 장치
- **Camera Component**를 설정해서 **Camera**를 만들 수 있다.
- 하나의 씬에 두개 이상의 **Camera**를 활용하여 구성 할 수 있다.



- **Clear Flags** : 화면의 빈공간의 처리를 어떻게 할 것인지 설정.
- **Background** : 모든 요소가 그려지고 스카이 박스가 없을 경우 여백의 색상
- **Culling Mask** : 카메라가 렌더링 할 오브젝트의 레이어를 포함하거나 제외한다. 오브젝트의 레이어를 **Inspector** 할당해야 한다.
- **Projection** : 카메라의 원근 **Simulation** 성능을 토글한다.
- **Size** : **Projection**에서 **Orthographic**을 선택하면 나타난다. 카메라의 사각형 크기를 나타낸다.
- **Field of view** : **Projection**에서 **Perspective**를 선택하면 나타난다. 로컬 **Y**축을 따라 측정한 카메라의 뷰 각도의 너비 입니다.
- **Clipping Planes** : 렌더링을 시작 및 중지하기 위한 카메라로 부터의 거리.
- **Viewport Rect** : 카메라 뷰가 드로우될 화면의 위치를 나타내는 네 개의 값을 의미한다.
- **Depth** : 드로우 순서의 카메라 포지션을 의미한다.
- **Rendering Path** : 카메라는 플레이어 설정에서 메서드를 정의하는 옵션.
- **Target Texture** : 카메라 뷰의 출력을 담을 **Render Texture**에 대한 **Reference** 이 설정을 하게 되면 카메라의 화면 렌더링 성능이 비활성화 된다.

2. CAMERA 설정과 PLAYER제작

주요 기능 상세 내용.



- **Viewport Rect**

1. **X** : 수평 위치의 시작점
2. **Y** : 수직 위치의 시작점
3. **W** : 화면상의 카메라의 출력 너비
4. **H** : 화면상 카메라의 출력 높이

- **Rendering Path**

1. **Use Player Settings** : 카메라는 플레이어 설정에서 설정한 렌더링 경로를 사용한다.
2. **Vertex Lit** : 카메라가 렌더링한 모든 오브젝트는 **Vertex-Lit** 오브젝트로 렌더링 된다.
3. **Forward** : 모든 오브젝트가 **Material**당 하나의 패스를 통해 렌더링 한다.
4. **Deferred Lighting** : 모든 오브젝트는 조명없이 드로우 되며, 그 후 모든 오브젝트의 조명이 렌더 대기열 끝에서 함께 렌더링된다. **Orthographic**에서는 항상 **Forward**이다.

- **Clear Flags**

1. **Skybox** : 여백을 스카이 박스로 채운다. **Skybox**가 설정되어 있지 않으면 자동으로 지정된 색상으로 채워진다.
2. **Solid Color** : 사용자가 지정한 색상으로 여백을 채운다.
3. **Depth Only** : 여백을 투명처리 한다. 이때 버퍼를 초기화해 기존에 그린 내용을 모두 제거한다.
4. **Don't Clear** : 버퍼를 초기화하지 않고 이전에 그려진 내용위에 바로 새로운 화면을 그린다.

- **Projection**

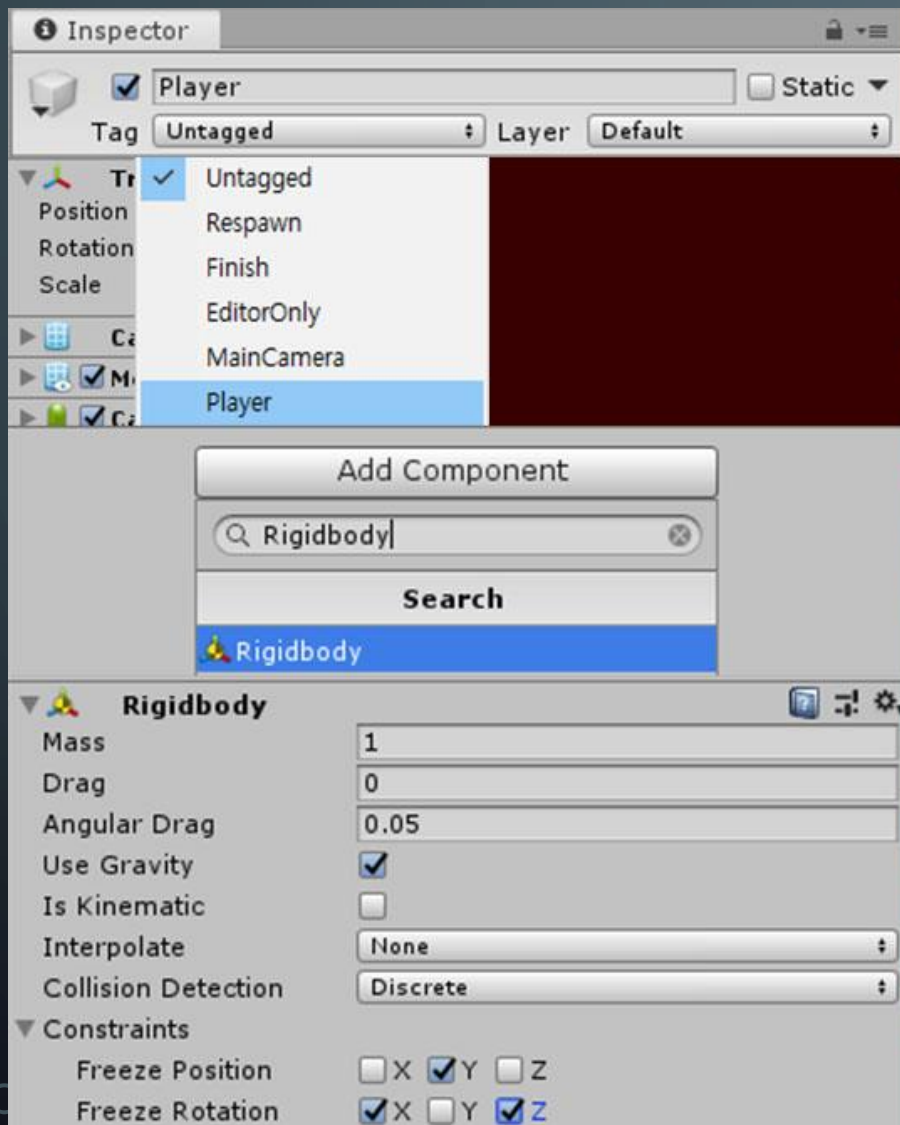
1. **Perspective** : 카메라가 원근감을 그대로 적용하여 오브젝트를 렌더링한다.
2. **Orthographic** : 카메라가 원근감이 없이 오브젝트를 균일하게 렌더링한다.

- **Clipping Planes**

1. **Near** : 드로잉이 수행될 카메라에 상대적으로 가장 가까운 포인트를 나타낸다.
2. **Far** : 드로잉이 수행될 카메라에 상대적으로 가장 먼 포인트를 나타낸다.

2. CAMERA 설정과 PLAYER제작

Player에 **Tag** 할당하고 **Rigidbody**를 추가하고 제약을 설정한다.



1. Player에 Tag를 설정한다, Inspector View에서 Popup List를 선택하면 적용할 Tag를 설정할 수 있다.

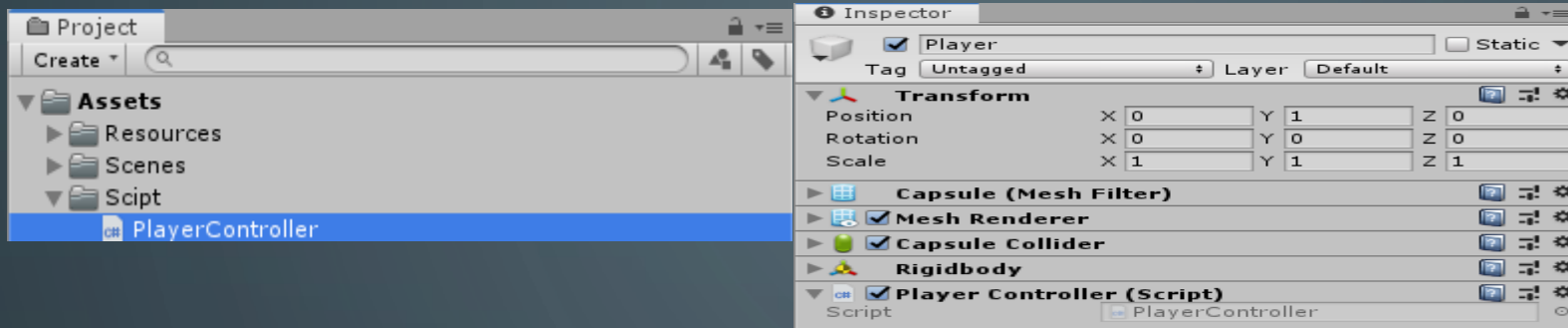
2. Player에 물리작용이 적용되기 때문에 Rigidbody를 Add Component해준다.

3. Rigidbody 속성중에 Constraints를 확장하면 물리작용에도 변화를 주지않고 고정시킬 방향축을 설정한다. 각각 Position과 Rotation에 적용할 수 있다.

* 기본적으로 물리 현상은 질량과 속도를 가진 물체가 물리작용을 했을때에 가속도와 질량과 방향에 의해 힘이 전달되게 되어 있다 그렇게 때문에 Player Object는 물리작용을 받는 Rigidbody를 가지고 있기 때문에 Collider에 의해 충돌을 판정하고 속도와 방향 질량등에 의해 힘이 가해지는 방향으로 회전하거나 밀려나거나 하는 현상이 벌어지기 때문에 물리작용을 받더라도 특정 상황에 벌어지지 않게 하는 역할을 하는 속성이다.

2. CAMERA 설정과 PLAYER제작

Player Script를 생성하고 **Rigidbody**를 사용할 수 있게 설정하자.

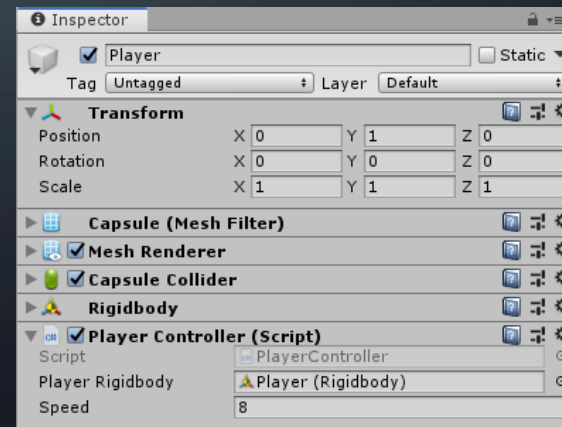


- **PlayerController**이라는 **Script**를 프로젝트의 원하는 경로에 생성하고 **PlayerController**를 **Player Object**에 **Add**하자.
- **PlayerController Script**를 편집하자. 물리작용의 코드 처리를 위해 **Rigidbody**를 할당 받을 준비를 하고 이동에 사용할 **Float Type** 변수를 만들어 놓자.
- **Public** 으로 선언되어 오픈한 **Type**들은 **Inspector View**에서 직접 설정할 수 있다 만약 특정 **Component**를 얻어오고 싶다면 **GetComponent**를 이용해도 된다.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public Rigidbody playerRigidbody;
    public float speed = 8f;

    void Awake()
    {
        playerRigidbody = GetComponent<Rigidbody>();
    }
}
```



PLAYER 입력감지

3. PLAYER 입력감지

Frame

- 컴퓨터 화면은 **1초에 60번** 정도 화면을 새로 그린다. 매번 새로 그리는 각각의 화면을 **Frame** 이라고 부른다.



- 1초** 동안 화면이 새로 그려지는 횟수를 초당 프레임(**FPS**)라 부른다.
 - PC나 콘솔게임의 화면은 보통 **60FPS**로 그려진다.
 - **60FPS**는 화면을 **1초에 60번** 갱신하므로 이전 프레임과 다음 프레임 사이의 시간 간격이 **1/60초**이다.
 - **60FPS**는 평균값일 뿐이고, 실제 **FPS**는 기기의 성능에 따라 달라진다.
- MonoBehaviour의 Update() Method가 매 Frame마다 실행하게 된다.**



3. PLAYER 입력감지

Input 클래스

사용자 입력을 감지하는 **Method**를 모아둔 클래스. → 실행 시점에 어떤 키를 눌렀는지 알려준다.
입력감지 **Method**를 **Update() Method**에서 실행하게 되면 매 **Frame**마다 입력을 체크할 수 있다.

```
void Update()  
{  
    //입력감지  
}
```

Update()가 초당 60번 실행
→ 입력 감지가 1/60초마다 실행

키를 누르고 있을 때
입력 감지가 실행되면 입력이 감지됨

PlayerController Script에 **Update() Method**에 사용자 입력을 감지하고 **playerRigidbody**에 힘을 가하는 코드를 작성하면 **Player Object**는 매 **Frame**마다 특정방향을 힘을 받아서 움직이게 된다.

```
if(Input.GetKey(KeyCode.UpArrow) == true)  
{  
    playerRigidbody.AddForce(0f, 0f, speed);  
}
```

원하는 키를 입력 받아서 힘을 가할 **Vector**를 적용하는 방법으로 **Player**를 이동시키고 있다.

Input.GetKey() Method는 해당 키를 ‘누르는 동안’ **true**, 그 외에는 **false** 반환
Input.GetKeyDown() Method는 해당 키를 ‘누르는 순간’ **true**, 그 외에는 **false** 반환
Input.GetKeyUp() Method는 해당 키를 누르다가 ‘놓는 순간’ **true**, 그 외에는 **false** 반환

KeyCode : 키보드의 키 식별자를 쉽게 가리키기 위한 타입이다. **KeyCode** 내부는 숫자로 동작한다.

3. PLAYER 입력감지

GameObject를 움직이는 또 다른 방법

transform.Translate() : GameObject가 **World Space** 공간에서 자신의 **Transform**의 좌표축이 어느 방향을 가리키고 있는지를 가지고 향하는 방향만큼의 **1**단위 크기로 이동변환을 시켜준다.

→ 이동크기를 변경하려면 **transform.forward** 값이 **1**단위 크기를 이동시키고 있으므로 **0.x**를 곱해주면 감소한다.

```
if (Input.GetKey(KeyCode.W))
{
    //Object의 정면으로 진행하는 방향으로 1크기 만큼 이동변환을 한다.
    transform.Translate(transform.forward, Space.World);
}
```

```
if (Input.GetKey(KeyCode.A))
{
    //Object의 오른쪽으로 진행하는 방향으로 -1크기 만큼 이동변환을 한다.
    transform.Translate(-transform.right, Space.World);
}
```

Rotate() : GameObject가 **World Space** 공간에서 자신의 **Transform**의 좌표축을 중심으로 **Angle**값을 받아서 회전 변환을 시켜준다.

```
if (Input.GetKey(KeyCode.Q))
{
    //Object의 Up Vector(y축) 방향을 축으로 반시계 방향으로 회전 변환한다.
    transform.Rotate(Vector3.up, -5, Space.World);
}
```

3. PLAYER 입력감지

Player Die 처리

Player Script에 **Die() Method**를 추가한다. → 외부에서 접근해서 실행하는 **Method**형태

```
public void Die()
{
    //자신의 GameObject를 비활성화
    gameObject.SetActive(false);
}
```

GameObject은 **Type**, **gameObject**는 변수

gameObject

Component 자신이 추가된 **GameObject**를 가리키는 변수이며 **MonoBehaviour** 에서 제공하고 있다.
모든 **Component**는 **gameObject** 변수를 이용해 자신을 사용 중인 **GameObject**에 접근할 수 있다.

3. PLAYER 입력감지

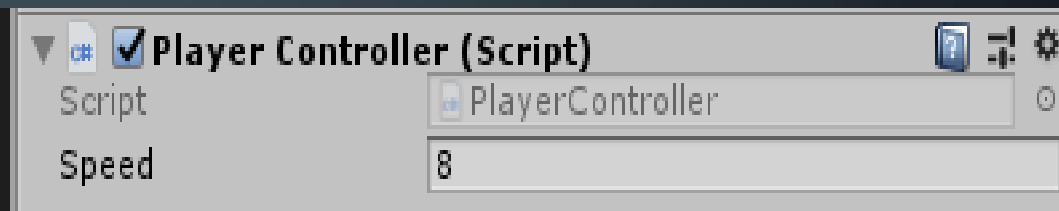
PlayerController Script 문제점 개선

- 조작이 게임에 즉시 반영되지 않는다.
 - **Rigidbody Component**의 **AddForce() Method**는 힘을 추가하는 방법이다. 누적된 힘으로 속도를 점진적으로 증가 시키기 때문에 속도가 충분히 빨라질 때까지 시간이 걸린다. 또한 이동 중에 반대 방향으로 이동하려는 경우 관성에 의해 힘이 상쇄되어 방향 전환이 금방 이루어지지 않는다.
- 입력 감지 코드가 복잡하다.
 - 방향키를 감지하는데 **if**문을 4개 사용하고 있다 좀 더 쉽고 간결한 코드로의 개선이 필요.
- playerRigidbody**에 **Component**를 **Drag&Drop**으로 할당하는 것이 불편하다.
 - **Drag&Drop**에 의한 할당방법은 **Rigidbody**가 여럿일때 문제가 될 수 있고 할당된 **Object**의 이름이나 할당 받는 타입의 변수명만 달라져도 해제가 되어서 다시 할당해줘야 한다.

playerRigidbody에 Component 할당개선.

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

```
void Start()  
{  
    //GameObject에서 Rigidbody 컴포넌트를 찾아 playerRigidbody에 할당  
    playerRigidbody = GetComponent<Rigidbody>();  
}
```



3. PLAYER 입력감지

GetComponent() Method

원하는 형식의 인스턴스를 가져다 달라는 요청을 하게 된다. 만약, 존재하지 않았다고 **Exception**을 발생하지 않으면 **null**을 전달해서 해당 형식이 없다고 알려준다.

조작감 개선하기

이미 설정되어 있는 수직과 수평 값을 체크해서 가속도를 적용하는 방법을 사용하였다.

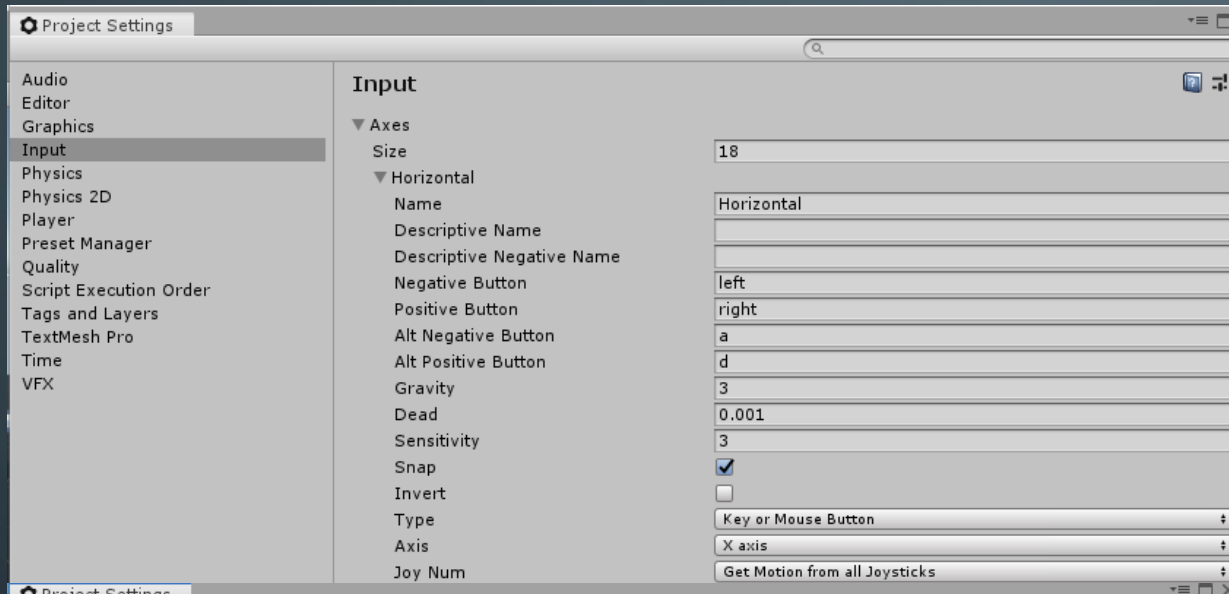
```
//수평축과 수직축의 입력값을 감지하여 저장
float xInput = Input.GetAxis("Horizontal");
float zInput = Input.GetAxis("Vertical");

//실제 이동속도를 입력값과 이동 속력을 사용해 결정
float xSpeed = xInput * speed;
float zSpeed = zInput * speed;

//Vector3 속도를 (xSpeed, 0, zSpeed)로 생성
Vector3 moveVelocity = new Vector3(xSpeed, 0f, zSpeed);
//리지드바디의 속도에 moveVelocity를 할당
playerRigidbody.velocity = moveVelocity;
```

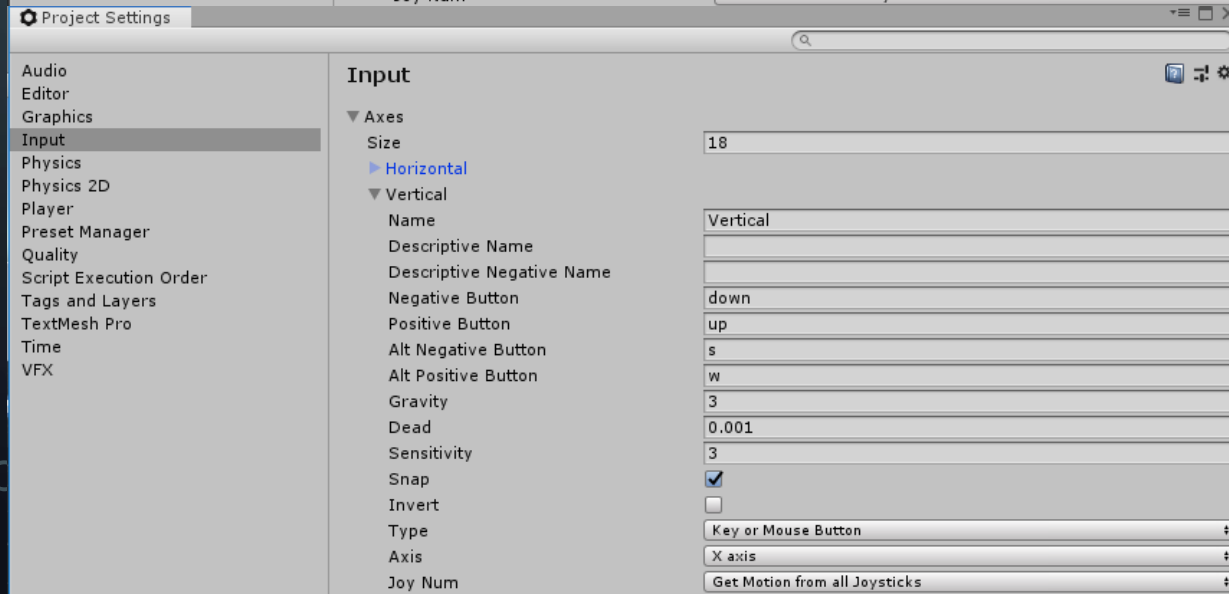
3. PLAYER 입력감지

Input 설정에 의해 설정된 조작과 체크를 통해서 값을 적용하고 있다.
Edit -> Project Setting -> Input에서 확인 할 수 있다.



Axes -> Horizontal에서 보면 입력키들을 각각 어떤 입력 커맨드와 연결할지 이미 설정되어 있기 때문에 **Input.GetAxis() Method** 를 사용할 수 있게 한다.

left, a 키는 음의 방향으로 (**-1.0**)
Right, d 키는 양의 방향으로 (**1.0**)
중력 가속도 **3**의 값으로 **0.001**이 되면 멈추는 것으로 설정되어 있다.



Axes -> Vertical에서 보면 입력키들을 각각 어떤 입력 커맨드와 연결할지 이미 설정되어 있기 때문에 **Input.GetAxis() Method**를 사용할 수 있게 한다.
down, s 키는 음의 방향으로 (**-1.0**)
up, w 키는 양의 방향으로 (**1.0**)
중력 가속도 **3**의 값으로 **0.001**이 되면 멈추는 것으로 설정되어 있다.

3. PLAYER 입력감지

GetAxis() Method

- **Input.GetKey()**를 이용해서 여러 번 키를 감시하는 대신 미리 준비된 **Axis** 옵션을 이용해 한번에 적용하였다.

```
public static float GetAxis(string axisName);
```

- 축의 이름을 **String**으로 받고 있다. **Horizontal, Vertical**중 하나를 **String**으로 넘겨주면 된다.

```
//수평축과 수직축의 입력값을 감지하여 저장  
float xInput = Input.GetAxis("Horizontal");  
float zInput = Input.GetAxis("Vertical");
```

- **xInput**과 **zInput**에 각각 축입력을 저장하게 된다.
- 그러므로 **xInput, zInput**에 각각 입력된 키에 따라 **1.0** 또는 **-1.0**이 입력된다.

```
//Vector3 속도를 (xSpeed, 0, zSpeed)로 생성  
Vector3 moveVelocity = new Vector3(xSpeed, 0f, zSpeed);  
//리지드바디의 속도에 moveVelocity를 할당  
playerRigidbody.velocity = moveVelocity;
```

- 속도 값을 보면 **Vector**로 되어 있다 여기서 입력된 값들이 적용되면 **Vector**가 향하는 방향으로 속도가 적용 될 것이라는 것을 알 수 있다. 원하는 환경에서 맞는 속도를 표현하기 위해 **speed** 값을 추가 적용해 보자.

```
//실제 이동속도를 입력값과 이동 속력을 사용해 결정  
float xSpeed = xInput * speed;  
float zSpeed = zInput * speed;
```


3. PLAYER 입력감지

AddForce와 velocity의 차이

관성에 의해서 **AddForce() Method**는 힘을 누적하고 속력을 점진적으로 증가 시켰고, **velocity**를 수정하는 것은 이전 속도를 지우고 새로운 속도를 대입 받는 것이기 때문에 관성을 무시하고 바로 적용 되는 것이다.

Project Chapter 4_1로 여기까지의 진행과정을 확인 할 수 있다.