Unity3D Basic

Input System

목차

- Unity Input
  - 키보드 입력(Keyboard input)
  - 키보드로 게임 오브젝트 이동

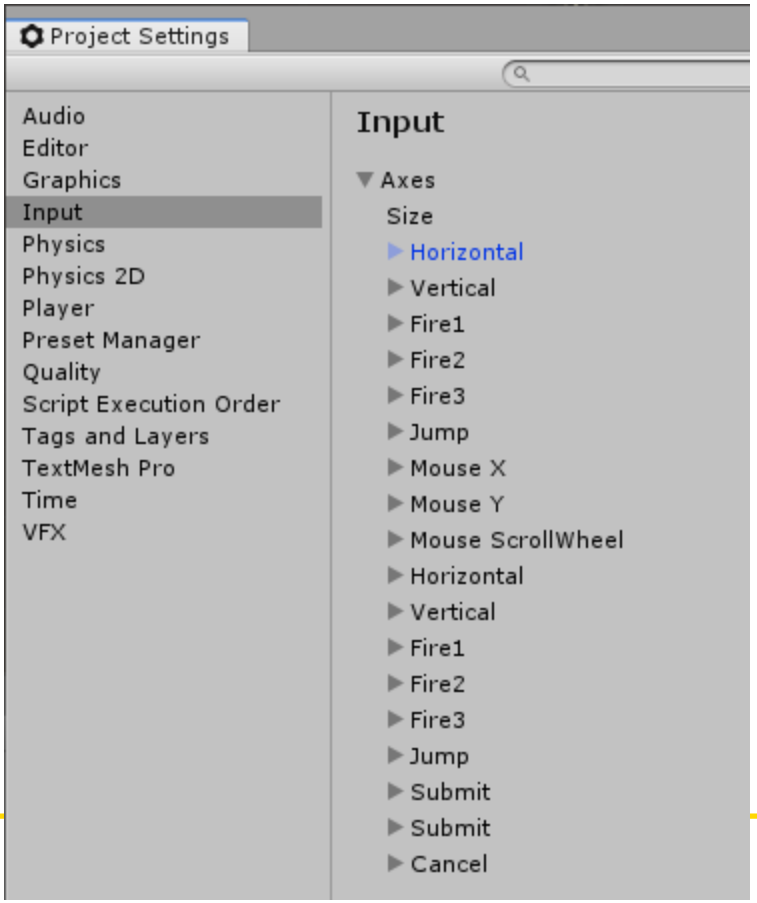
- **키보드 입력**

  1. 미리 설정한 키 조합(버튼) 사용
  2. 특정 키 입력

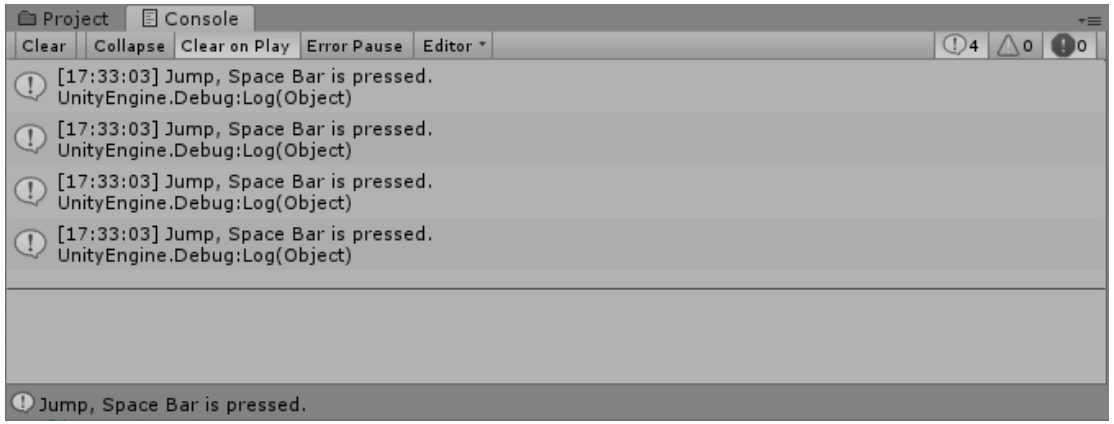
(1) 미리 설정한 키 조합(버튼) 사용

- [Edit – Project Setting – Input] 에서 미리 설정된 키 조합 사용 (수정가능)
- GetButton(), GetButtonDown(), GetButtonUp()
- 버튼으로 설정된 이름으로 동작

(예) if (Input.GetButton("Fire1")) { ... }



```
public class InputTest : MonoBehaviour
{
    void Update() {
        if (Input.GetButtonDown(“Jump”)) {
            Debug.Log(“Jump, Space Bar is pressed.”);
        }
    }
}
```



GetButton(), GetButtonDown(), GetButtonUp() 을 차례로 실행해 보세요

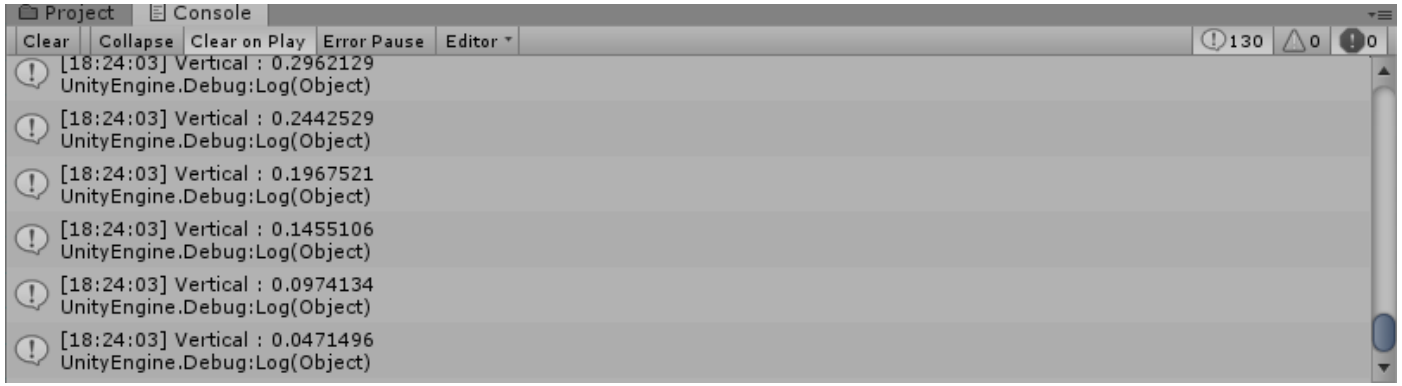
(1) 미리 설정한 키 조합(버튼) 사용

- `Input.GetAxis(“키이름”)`
  - “키 이름"에 해당하는 키보드를 누르면 -1.0f ~ 1.0f 까지의 값을 반환하고, 누르지 않으면 0.0f 를 반환
  - `GetAxis()` : -1 ~ 1 사이 값을 반환 (부드러운 이동이 필요할 때)
  - `GetAxisRaw()` : -1, 0, 1 값을 반환 (즉시 반응해야 할 때)

(예) `h = Input.GetAxis(“Horizontal”);`

- Horizontal 에 해당하는 키를 누를 시, -1.0f ~ 1.0f 까지의 값을 반환
- 오른쪽 화살표(또는 D)를 누르면, 0~1 사이의 값을, 왼쪽 화살표(또는 A)를 누르면 -1~0 사이의 값을 리턴

```
public class InputTest : MonoBehaviour
{
    float v;

    void Update() {
        v = Input.GetAxis("Vertical");
        if ( h != 0.0 ) {
            Debug.Log("Vertical : " + v);
        }
    }
}
```



GetAxis(), GetAxisRaw() 을 차례로 실행해 보세요

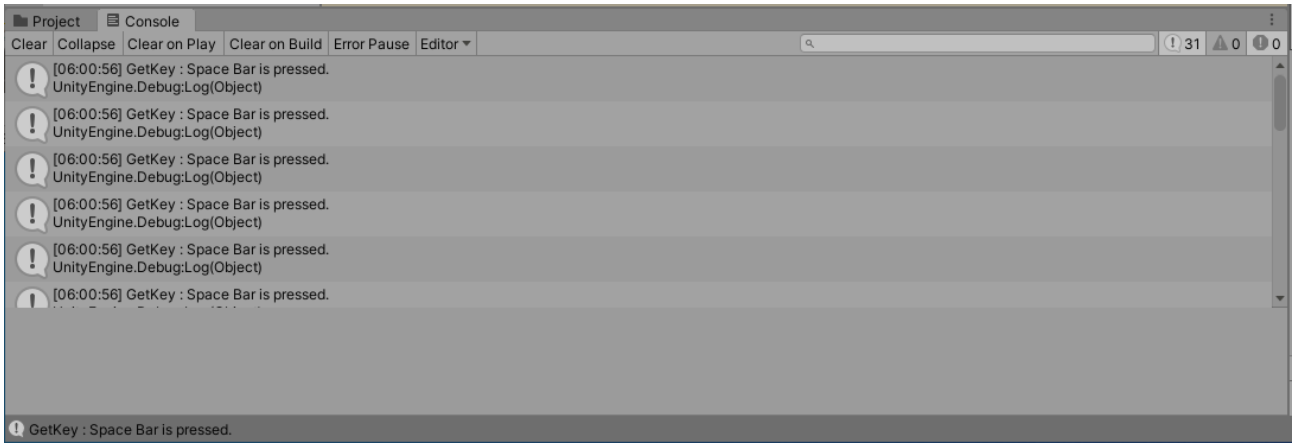
(2) 특정 키 입력

- GetKey(), GetKeyDown(), GetKeyUp()
- 키보드의 키코드를 인식하여 동작
  - » 키 입력은 [KeyCode.특정키]
  - » 방향 키는 UpArrow, DownArrow, RightArrow, LeftArrow

(예) `if (Input.GetKeyDown(RightArrow)) { ... }`

  - » 오른쪽 방향키를 한번만 눌러 실행하고자 할 때

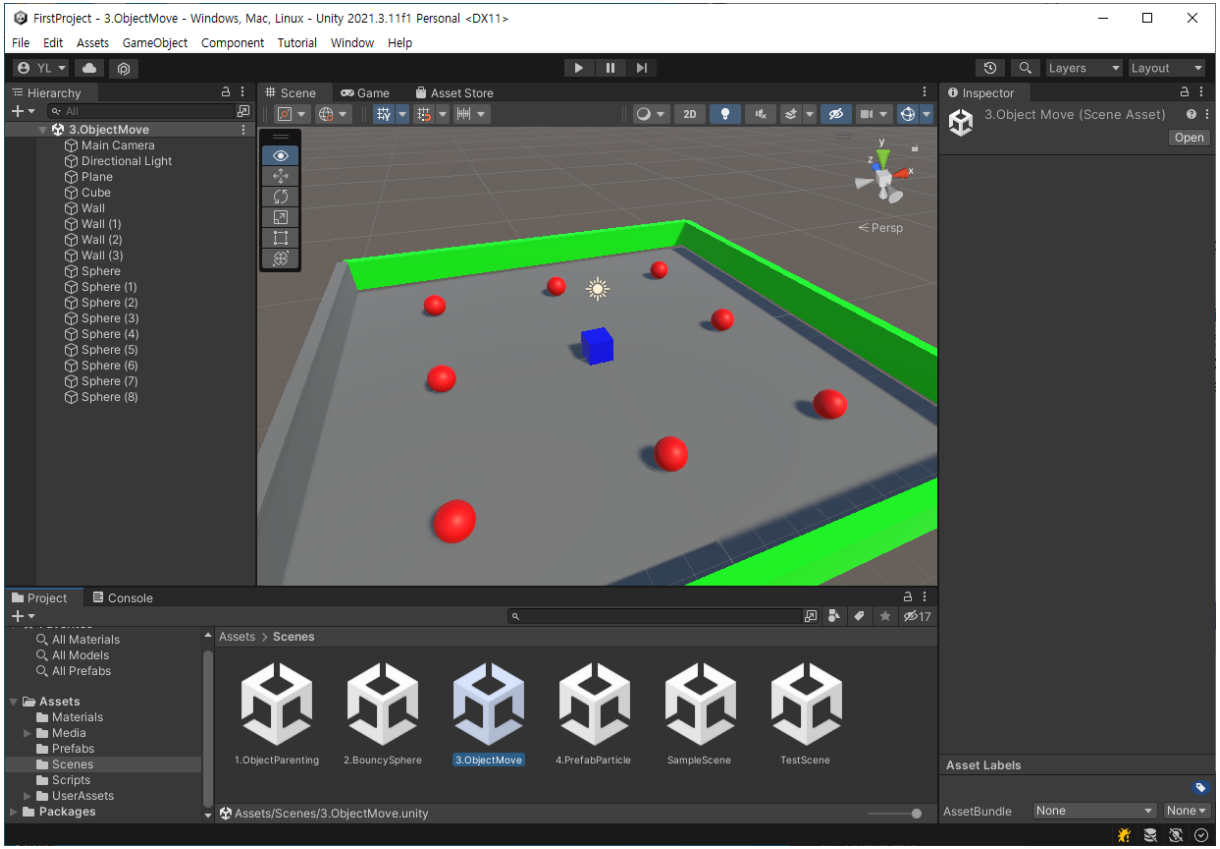
```
public class InputTest : MonoBehaviour
{
    void Update() {
        if (Input.GetKeyDown(KeyCode.Space)) {
            Debug.Log("Getkey : Space Bar is pressed.");
        }
    }
}
```



GetKey(), GetKeyDown(), GetKeyUp() 을 차례로 실행해 보세요

게임 오브젝트 이동 및 충돌

- 키보드로 게임오브젝트 이동



1. 이동 스크립트를 작성하여 Cube 오브젝트에 드래그
2. 게임 실행 모드에서 키보드의 화살표 키를 입력으로 오브젝트 이동 여부를 확인
3. Wall 과 Sphere 오브젝트를 다수 개 생성하여 배치
4. Cube 오브젝트에 리지드바드를 추가하고 이동하면서 Wall 과 Sphere 오브젝트와 충돌 실험

   (실행 모드에서 Wall/Sphere 오브젝트 Collider 속성의 IsTrigger 를 체크하면서 충돌과 관통을 실험)
5. 충돌 스크립트를 작성하여 Cube 오브젝트에 드래그 (또는 이동스크립트에 추가)

   (충돌을 인지하면, 충돌 오브젝트를 제거 – 예로, 아이템을 획득하면 획득 아이템은 화면에서 제거)
6. 충돌 오브젝트에 Tag 를 부여하여 (특정) 대상만을 구분하여 제거함

– 게임 오브젝트를 이동시키는 방법

1) transform의 Position에 직접 접근하여 (값을) 설정해주는 방법
2) transform의 Translate() 함수를 이용하는 방법
3) RigidBody를 설정한 후, velocity(속도) 값을 설정해주는 방법 (물리 시뮬레이션에서 주로 사용하는 방법)

- (1번) (2번)은 transform을 이용하는 방법으로, 물리 법칙이 적용되지 않는 대상일 때 사용
  - transform을 이용해 강제적으로 이동하면, 물리 법칙을 무시하고 강제로 절대 위치를 바꿔버리는 것이기에 위치가 옮겨진 후 물리법칙에 문제가 발생할 수 있음
- (3번)은 Rigidbody 컴포넌트를 이용하는 방법으로, 물리 법칙이 적용되어 있는 대상일 때 사용
  - 물리가 적용되어 있는 Rigidbody 이기 때문에, Rigidbody 컴포넌트를 이용하여 이동하는 것이 좀 더 자연스러운 물리 법칙이 적용될 수 있음

– 게임 오브젝트를 이동시키는 방법

1) transform의 Position에 직접 접근하여 (값을) 설정해주는 방법

```
transform.position += new Vector3(0,0,1);
```

2) transform의 Translate() 함수를 이용하는 방법

```
Transform.Translate(new Vector3(0,0,1));
```

3) RigidBody를 설정한 후, velocity(속도) 값을 설정해주는 방법

```
// RigidBody를 설정한 후, 속도값을 설정
RigidBody rigidBody = gameObject.GetComponent<Rigidbody>();
rigidBody.velocity = new Vector3(0,0,1);
// RigidBody에 Force를 적용하여 움직임
rigidBody.AddForce(new Vector3(0,0,1));
```

4) Character Controller를 이용하여 움직이는 방법

```
CharacterController charController =
gameObject.GetComponent<CharacterController>();
charController.Move(new Vector3(0,0,1));
```

– 이동 방향과 Vector

- 3차원 공간에 있는 오브젝트의 위치나 방향을 나타내려면 (x,y,z) 각 축의 값이 필요함
- 유니티에서는 이 값을 Vector3(x,y,z)로 표시하며, 인스팩터의 Position, Rotation, Scale 값이 이에 해당됨
- 2D 게임에서는 z축을 사용하지 않으며, Vector2(x,y)로 표시

  RGBA 칼라를 표시할 때는 Color(r,g,b,a) 또는 Vector4(r,g,b,a) 형식으로 사용
- Vector는 오브젝트의 방향 표시에 자주 사용되며, (x,y,z) 값이 각각 0 또는 1로 표시하고 반대방향은 음수로 지정함

– 플레이어가 움직임을 제어하기 위해, 키보드를 통해 게임 오브젝트를 이동

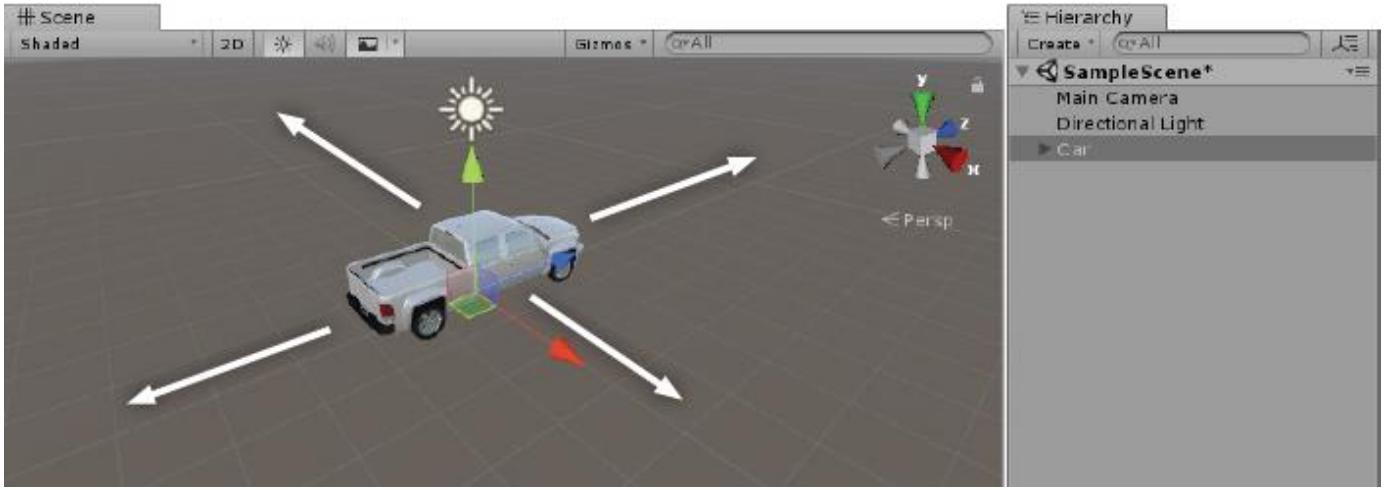
  키보드로부터 입력 받은 값을 새로운 방향 벡터로 설정


```
public class ObjectMove : MonoBehaviour
{
    float moveSpeed = 10f;  // 이동속도

    // Start is called before the first frame update
    void Start() { }

    // Update is called once per frame
    void Update()
    {
        // 현재 프레임에서 이동할 거리
        float amount = moveSpeed * Time.deltaTime;

        // 전후(Vertical) 좌우(Horizontal) 이동키를 받음
        float vert = Input.GetAxis("Vertical");
        float horz = Input.GetAxis("Horizontal");

        // 오브젝트의 전방으로 이동
        transform.Translate(new Vector3(horz,0,vert) * amount);    // 상대좌표
    }
}
```

Input.GetAxis() 함수는 키보드, 마우스 등의 입력 신호를 -1.0~1.0 사이의 아날로그적인 값으로 구하는 함수

게임을 실행하여 동작 여부를 확인

- 키의 반응은 게임뷰에서 동작



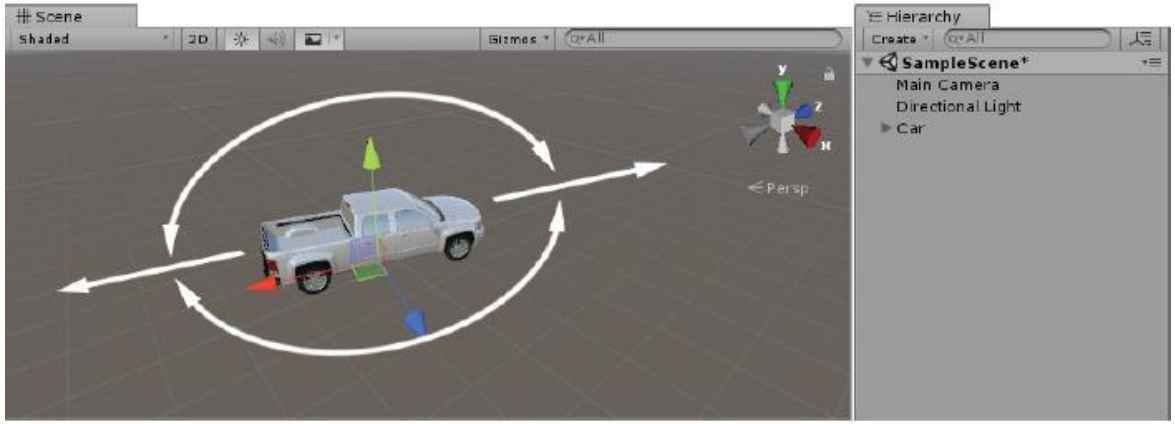
– 게임 오브젝트에 따라, 움직임이 이동 일수도 있지만, 회전 일수도 있음

: 전후 이동과 좌우 회전으로 스크립트를 수정


```
public class ObjectMove : MonoBehaviour
{
    float moveSpeed = 10f;  // 이동속도
    float rotateSpeed = 60f;   // 회전속도(초속 60)

    // Start is called before the first frame update
    void Start() { }

    // Update is called once per frame
    void Update()
    {
        // 현재 프레임에서 이동할 거리
        float amount = moveSpeed * Time.deltaTime;
        float amountRotate = rotateSpeed * Time.deltaTime;

        // 전후(Vertical) 좌우(Horizontal) 이동키를 받음
        float vert = Input.GetAxis("Vertical");
        float horz = Input.GetAxis("Horizontal");

        // 오브젝트의 전방으로 이동 (전진)
        transform.Translate(Vector3.forward * amount * vert);

        // 좌우회전
        transform.Rotate(Vector3.up * amountRotate * horz);
    }
}
```

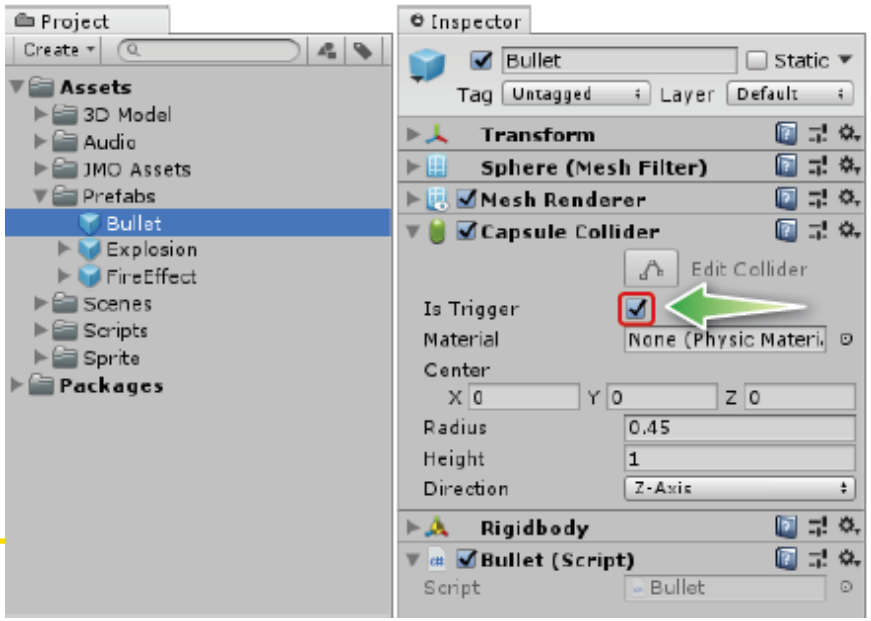
Transform.Rotate() 함수는 오브젝트를 회전시키도록 하며, 회전 축에 각도를 곱하여 회전각을 얻는다

스크립트를 수정한 다음, 게임을 실행



충돌의 판정과 처리

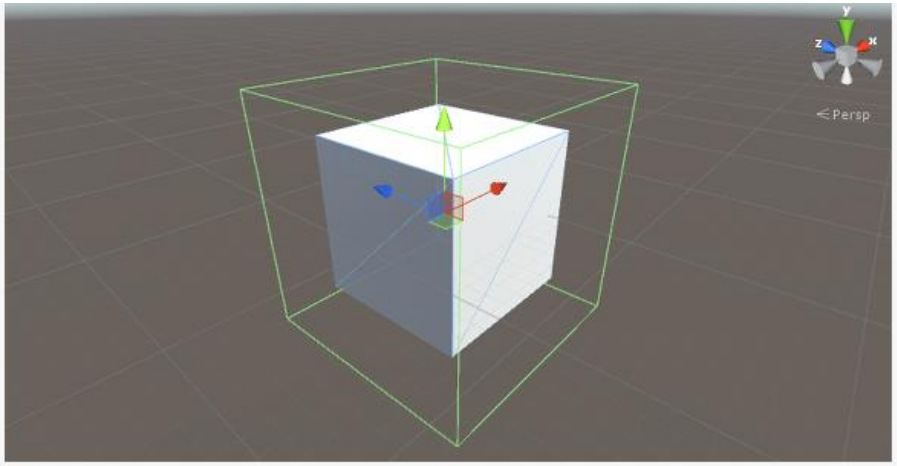
- 게임에서 충돌 판정 및 처리는 매우 중요
- 충돌 이벤트는 Rigidbody가 있는 오브젝트에서만 발생
  - 충돌 이벤트의 종류
    - 충돌이 발생할 때, 충격을 가하는 오브젝트가 반사되느냐 관통하는냐에 따라 발생하는 이벤트가 달라지며, 반사 및 관통 여부는 Collider의 Is Trigger 옵션으로 설정
      - Is Trigger ON : 오브젝트 관통, Trigger 이벤트 발생
      - Is Trigger OFF : 오브젝트에서 반사, Collision 이벤트 발생



- **충돌** : 2개의 오브젝트가 서로 접촉하는 상태
- 적어도 어느 하나가 Trigger On 이면 Trigger 이벤트가, 둘 다 Off면 Collision 이벤트가 발생한다
- 충돌 이벤트의 매개변수의 type
  - OnTrigger … (Collider other) Trigger 이벤트
  - OnCollision ... (Collision other) Collision 이벤트
- Collision의 반사는 절대적인 것이 아니라서 작은 물체가 빠른 속도로 이동하는 경우에는 가끔씩 오브젝트를 관통하기도 한다. 총알이나 화살 같은 발사체는 오브젝트를 관통하기도 하고 반사되기도 하므로, 일반적으로 Trigger를 On을 설정

– 충돌 처리를 위한 조건

- 충돌이 일어나기 위해서는 두 GameObject가 모두 Collider를 가지고 있어야 하며, 둘 중 하나는 Rigidbody를 가지고 있어야 한다
- 두 GameObject 중 하나만 움직인다면, 움직이는 GameObject가 Rigidbody를 가지고 있어야 한다



- 위 그림은 Collider를 잘 보이게 하기 위해 GameObject 보다 크기를 약간 키워 보임. 위에서 보이는 초록색 부분이 Collider로 실제로 충돌을 감지하는 영역임
- 유니티에서 제공하는 Object들에는 기본적으로 Collider가 들어가 있으며, Box Collider, Sphere Collider, Capsule Collider 등이 있음. 다른 Model을 불러와 작업한다면, Model에 알맞게 Collider를 설정해 줘야 올바른 충돌 처리를 할 수 있음

- Trigger

Trigger는 GameObject간의 물리적 연산을 하지 않고 충돌을 감지할 수 있다. 즉, 두 GameObject가 접촉했을때 서로 튕겨 나가지않고 그냥 통과하게 된다.

Trigger를 쓰기 위해서는 해당 Collider의 Is Trigger 항목을 체크해야 한다.

다음으로 스크립트에서 함수로 충돌 이벤트를 처리할 수 있다.

```
1 void OnTriggerEnter(Collider col) { }
2 void OnTriggerStay(Collider col) { }
3 void OnTriggerExit(Collider col) { }
```

다음과 같이 Enter, Stay, Exit 3가지 버전의 OnTrigger 함수를 만들 수 있다.
3가지 함수는 다음과 같은 특징을 나타낸다.
Enter - 충돌이 시작되는 순간 호출
Stay - 충돌이 되고있을 때 매 프레임마다 호출
Exit - 충돌이 끝날 때 호출

함수의 파라메터로 Collider 객체가 들어오며 col을 이용해 충돌한 GameObject에 대한 처리를 할 수 있다.

- Collision

Collision은 Trigger와 다르게 물리적인 연산을 하며 충돌을 감지한다.
주의할 것은 Rigidbody의 Kinematic 속성이 꺼져 있어야 충돌이 발생할 수 있다.

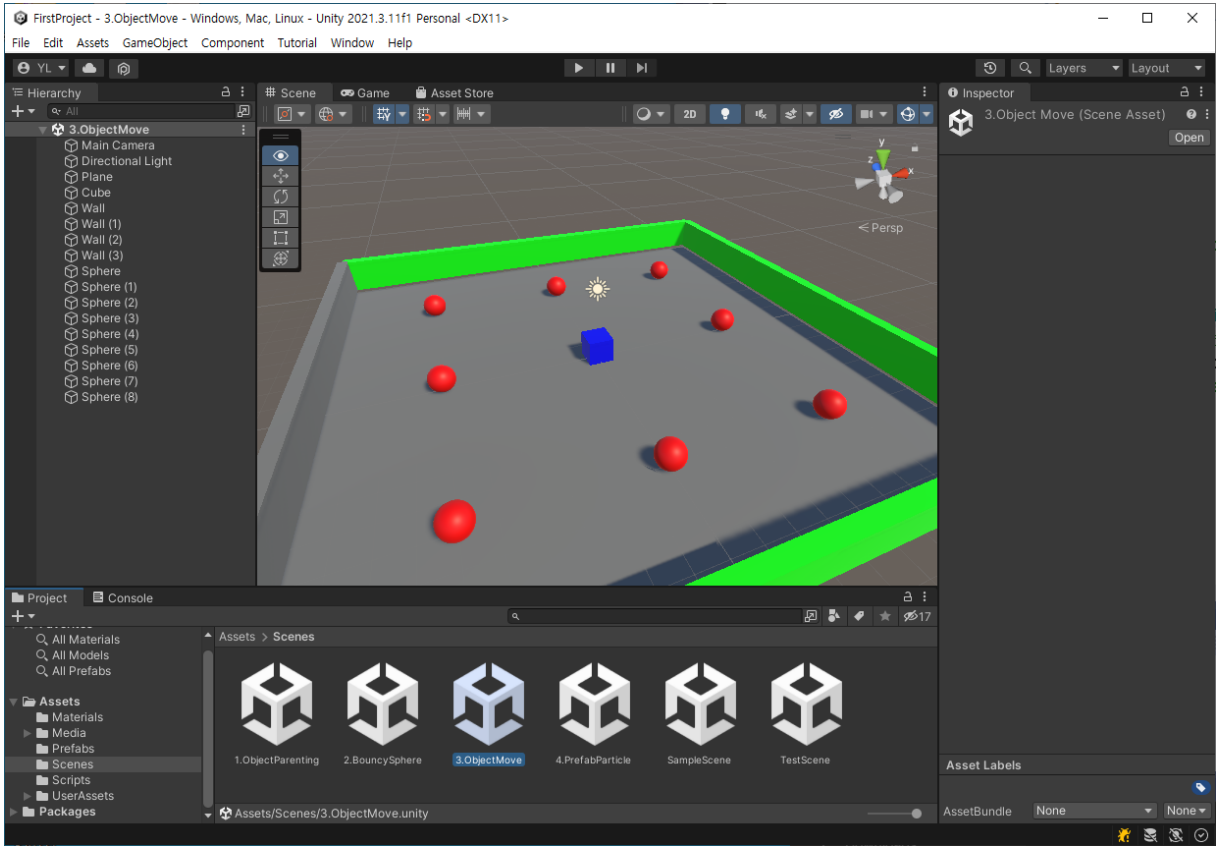
```
1 void OnCollisionEnter(Collision col) { }
2 void OnCollisionStay(Collision col) { }
3 void OnCollisionExit(Collision col) { }
```

Trigger와 동일하게 3가지 함수를 만들 수 있으며 특징 또한 동일하다.

함수의 파라메터로 Collision 객체가 들어오며 col을 이용해 충돌한 GameObject에 대한 처리를 할 수 있다.

게임 오브젝트 이동 및 충돌

- 게임오브젝트 충돌 판정 및 처리



1. Sphere 오브젝트를 프리팹으로 만듬
2. (씬에 존재하는) Sphere 오브젝트는 제거하고 모두 프리팹으로 대체함

   ★ 프리팹을 사용하는 장점

   프리팹을 수정하는 경우 vs. 씬에 있는 오브젝트를 수정하는 경우
3. 충돌 시, SendMessage()를 이용하여 충돌 처리 과정을 충돌 오브젝트의 스크립트로 분리
4. 충돌 오브젝트(아이템)을 제거하면서 파티클 효과를 추가

   파티클은 AssetStore 에서 무료 애셋을 검색하여 사용 (Unity Particle Pack 5.x)
5. 파티클이 계속 유지되고 있을 경우, 파티클 오브젝트 제거

– Trigger 이벤트 또는 Collision 이벤트 발생 여부 확인

```
public class ObjectMove : MonoBehaviour
{
    float moveSpeed = 10f;  // 이동속도

    // Start is called before the first frame update
    void Start() { }

    // Update is called once per frame
    void Update() { ... }

    private void OnTriggerEnter(Collider other) {
        Debug.Log("Trigger event is occurred..");
    }

    private void OnCollisionEnter(Collision collision)  {
        Debug.Log("Collision evernt is occurred");
    }
}
```

대상 객체(벽)에서 isTrigger 속성을 변경하여 발생하는 이벤트를 확인해 보세요

– 충돌 처리

- 충돌 오브젝트에 따라 처리 과정이 달라짐


```
public class ObjectMove : MonoBehaviour
{
    float moveSpeed = 10f;  // 이동속도

    // Start is called before the first frame update
    void Start() { }

    // Update is called once per frame
    void Update() { ... }

    private void OnTriggerEnter(Collider other) {
        Debug.Log("Trigger event is occurred..");
    }

    private void OnCollisionEnter(Collision collision)  {
        Debug.Log("Collision evernt is occurred");

        // 충돌 처리
        if (collision.gameObject.tag == "Item")
        {
            // 파괴(제거)
            //Destroy(collision.gameObject);
            // 안보이게
            collision.gameObject.SetActive(false);
        }
    }
}
```

– 충돌 처리

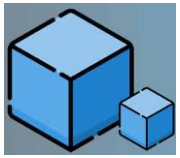
- 충돌 이벤트는 Rigidbody가 있는 오브젝트에서만 발생하므로, 벽이나 Target 은 충돌 이벤트가 발생하지 않음
- 모든 충돌에 대한 처리를 하나의 오브젝트에서만 처리하게 되면, Target 종류가 많아질수록 처리과정이 복잡해 질 수 있음 → **(움직이는) 오브젝트는 충돌 판정만 하고, 세부적인 처리는 Target 오브젝트에서 하는 것이 보다 효율적임**
- Target 오브젝트의 스크립트
  - TargetDestroy 스크립트를 작성하고 프리팹으로 만든 Target 오브젝트로 스크립트를 연결 (프리팹을 변경하므로, 씬 내의 모든 Target 오브젝트에 함께 반영됨)

```
public class TargetDestroy : MonoBehaviour
{
    .....

    void DestroySelf(Vector3 pos)
    {
        Destroy(gameObject);
    }
}
```

– 충돌 처리

- 수정


```
public class ObjectMove : MonoBehaviour
{
    ...
    private void OnCollisionEnter(Collision collision)  {
        ...

        // 충돌이 발생한 오브젝트가 아이템(Item)이면
        if (collision.gameObject.tag == "Item") {
            // (3) 오브젝트에서는 충돌 판정만 하고,
            // 충돌에 따른 세부처리는 충돌 대상 오브젝트에서 처리하는 것이 효율적임.
            // (충돌 대상 오브젝트는 collision.gameObject)
            collision.gameObject.SendMessage("DestroySelf", transform.position);
        }
    }
}
```



```
public class TargetObjectCollision : MonoBehaviour
{
    ...
    void DestroySelf(Vector3 pos) {
        //Debug.Log("DestroySelf method is called...");

        // 사운드(음향효과), 파티클, 점수 획득 등 (추가적으로 처리해줘야 할 내용)
        // 객체 제거
        //Destroy(gameObject);
        // 오브젝트 제거 대신에, 오브젝트가 화면에 안보이게
        gameObject.SetActive(false);
        }
    }
}
```

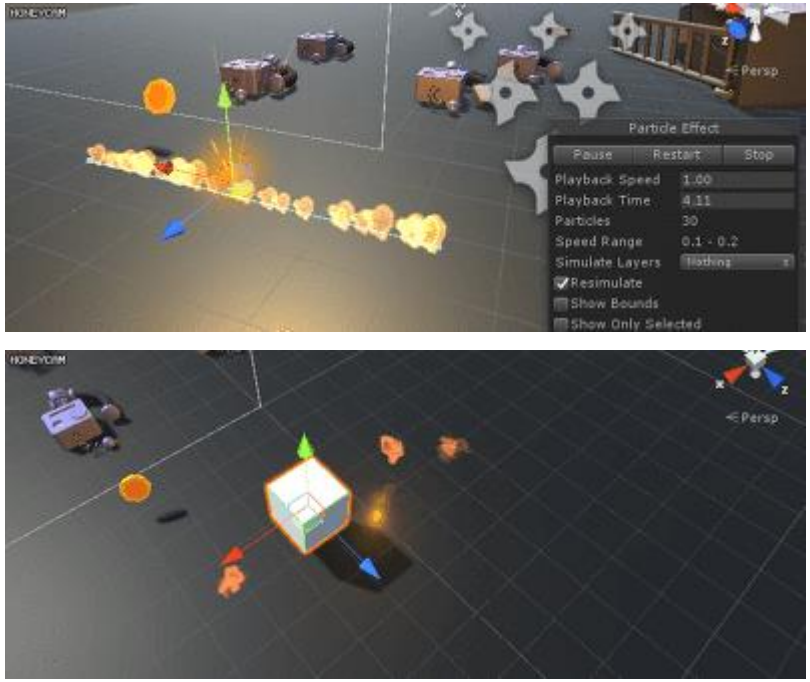
- 충돌 오브젝트(Target)에서 충돌 발생에 따른 처리
  - 파티클(Particle) 다운로드, 임포트 (AssetStore에서 “Particle Pack”)
  - GameManager 객체 생성, 스크립트 연결


```
public class TargetObjectCollision : MonoBehaviour
{
    // 오브젝트 제거 파티클
    public Transform explosion;

    // Start is called before the first frame update
    void Start() {}
    // Update is called once per frame
    void Update() {}

    // Object Destroy
    void DestroySelf(Vector3 pos) {
        // 오브젝트 제거 시점에서 파티클 실행
        //Instantiate(explosion, pos, Quaternion.identity);
        // 파티클 오브젝트 생성 후, 삭제되지 않는다면 이렇게
        Transform parti = Instantiate(explosion, pos, Quaternion.identity);
        // Delay(지연)을 줘서 오브젝트 제거(destroy)
        Destroy(parti.gameObject, 2);

        //Destroy(gameObject);
        gameObject.SetActive(false);

        // 아이템 획득 수(score)를 올려줌
        GameManager.AddResource();

    }
}
```

```
public class GameManager : MonoBehaviour
{
    [HideInInspector]
    public static int resource;

    // Start is called before the first frame update
    void Start() { resource = 0; }

    // Update is called once per frame
    void Update() {
        Debug.Log("Score : " + resource.ToString());
    }

    public int GetResource() { return resource; }
    public static void AddResource(int addCount) { resource += addCount; }
    public static void AddResource() { resource++; }
}
```

Particle System

- 파티클 시스템
  - 파티클이라는 매우 작은 이미지나 메쉬를 시뮬레이션하고 랜더링하여 시각효과를 생성
  - 불, 연기, 액체 등과 같은 동적 오브젝트를 구현할 때 유용



– Target의 폭파 불꽃

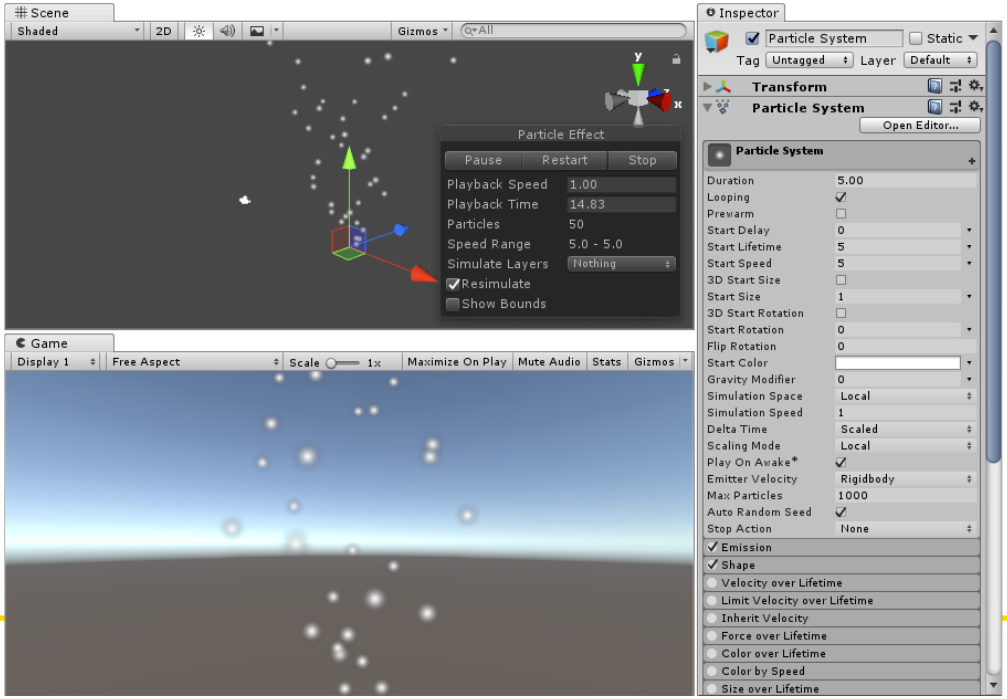
• Target이 폭파될 때의 화려한 폭파 불꽃을 만듦

(1) 파티클(Particle) 만들기

– 메뉴 [GameObject - Effects – Particle System]를 클릭하여 파티클을 추가

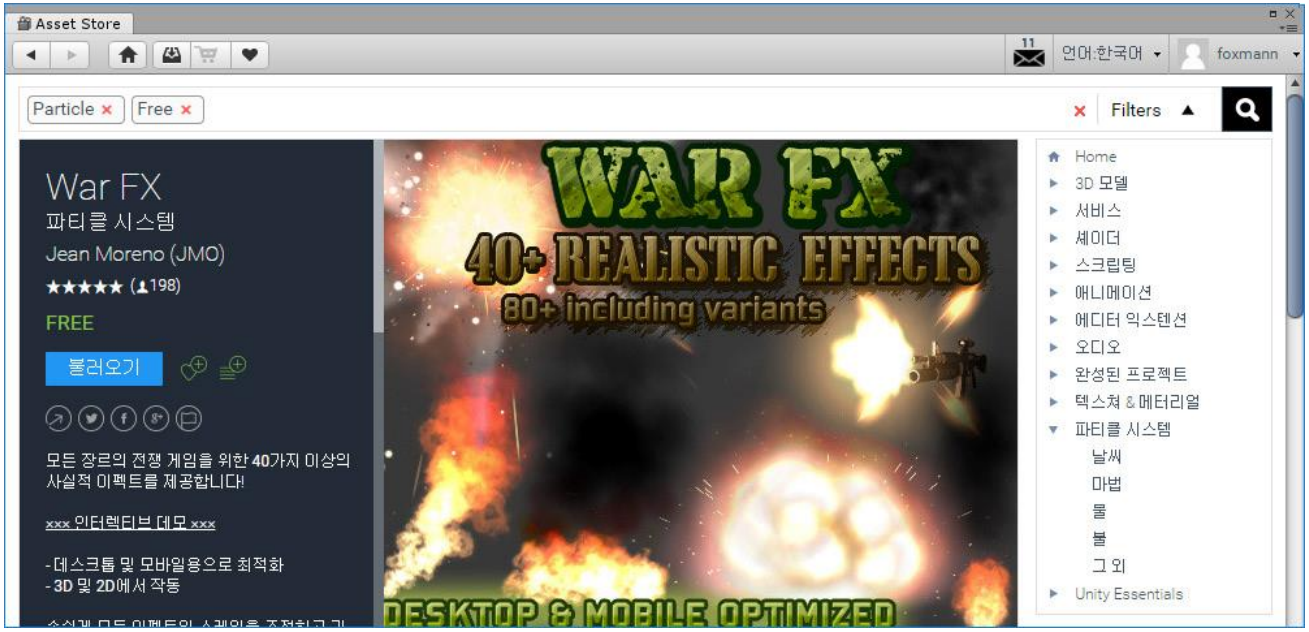
– 눈송이 같은 입자가 여기저기 날아감

– 게임 개발 측면에서 보면, 파티클은 디자인 영역이므로, 개발자가 직접 만드는 것은 권장하지 않음

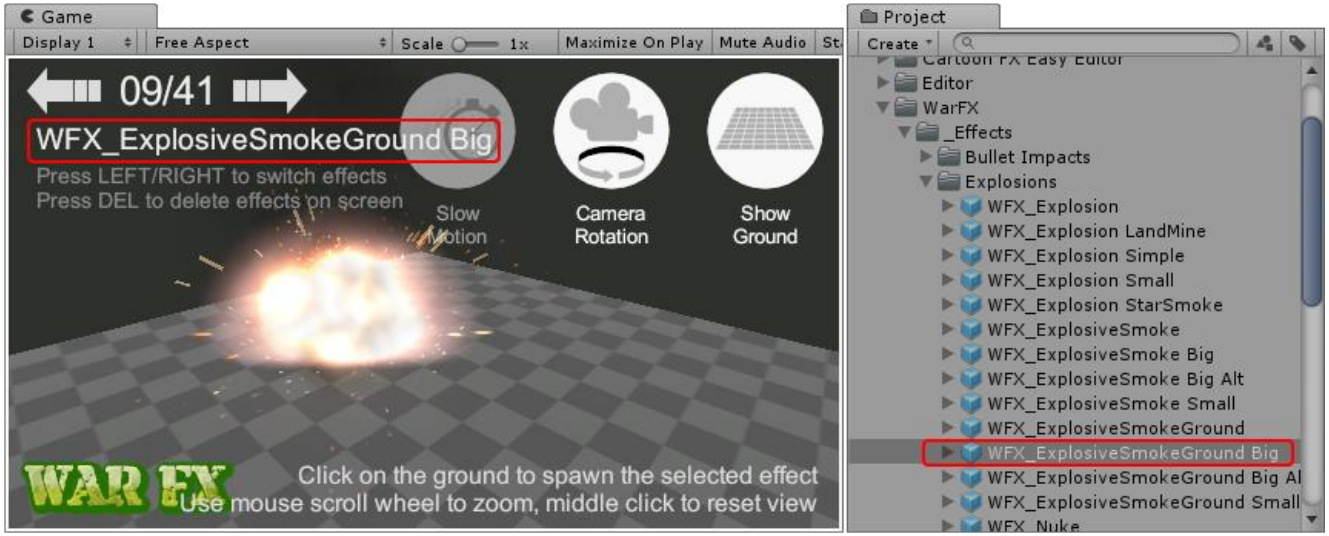


(2) Particle Asset 추가

– 에셋 스토어에서 “Particle Free”로 검색하여 “War FX”라는 무료 파티클 에셋을 다운로드



- 추가한 에셋은 프로젝트의 JMO Assets 폴더에 저장됨
- 40개 이상의 파티클이 있으며, 자신의 프로젝트에 필요한 것을 고름
- War FX/Demo 폴더의 데모 씬을 실행하고, 바닥을 클릭하면 파티클이 표시됨
- 게임 뷰 위쪽의 좌우 화살표는 이전/이후의 파티클을 표시
- 여러 파티클을 실행하여 적당한 것을 고름



(3) Particle을 프리팹 폴더에 복사

- War FX는 파티클이 프리팹으로 되어 있음
- 게임 뷰에 프리팹 이름이 표시됨
- 사용하려는 파티클을 _Effects 폴더에서 찾아 Prefabs 폴더로 복사하고, 이름을 Explosion으로 설정 (프로젝트 뷰에서는 Ctrl-D로 복제한 후, 복제한 파티클을 Prefabs 폴더로 드래그하고 해당 씬에서 적용함)

- 파티클을 프리팹으로 만들어 놓고, 충돌 처리에서 게임 객체를 제거할 때 파티클을 게임 뷰에 표시한다

```
Instantiate(explosion, transform.position, Quaternion.identity);
```

  - » Quaternion.identy는 회전각을 쿼터니언으로 표시한 것으로, identity는 회전하지 않은 각도

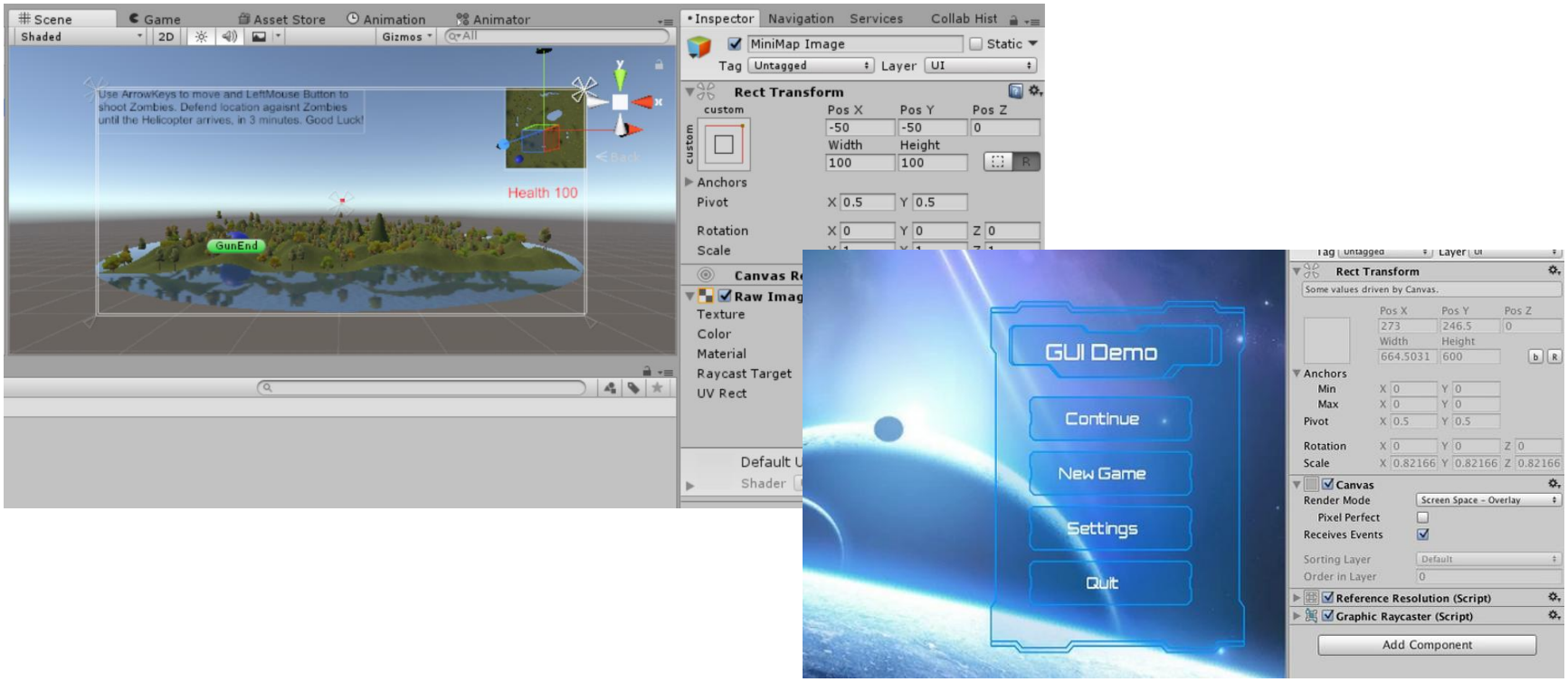
Canvas

캔버스 다루기

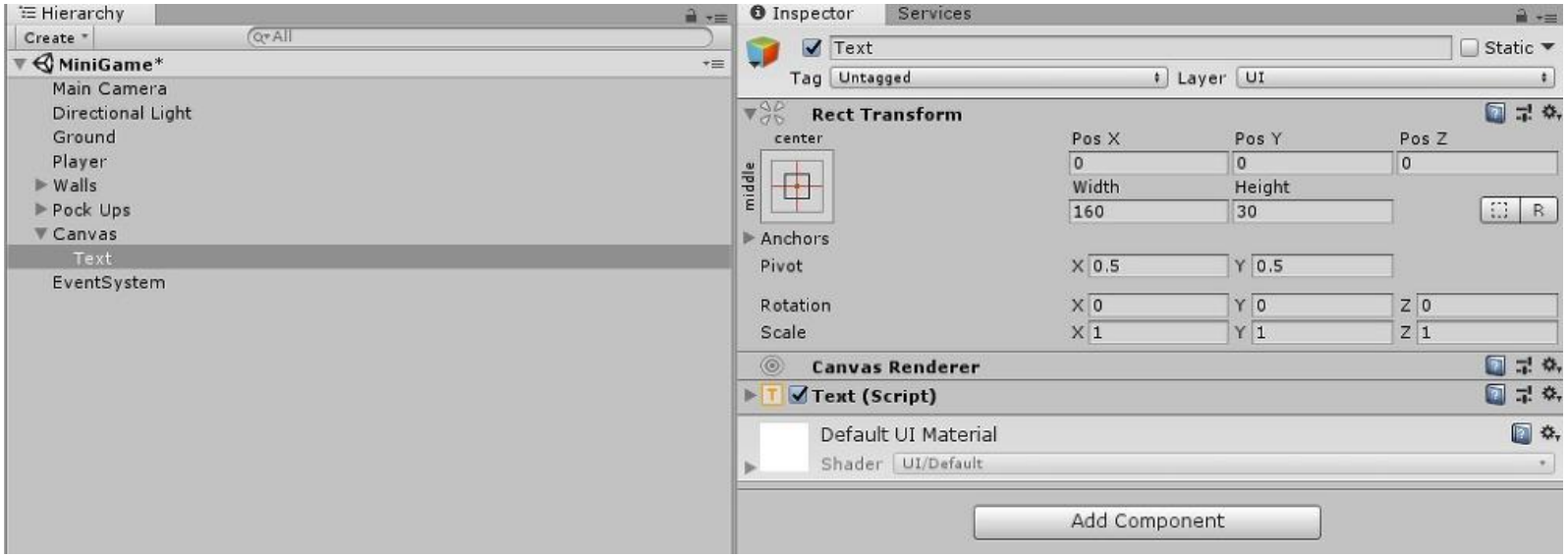
- 하이러리키 창에서 [Create – UI – Canvas]를 클릭하여 캔버스를 추가

  (Canvas와 함께 Event System이 추가됨 – 여기서는 Canvas만 사용)

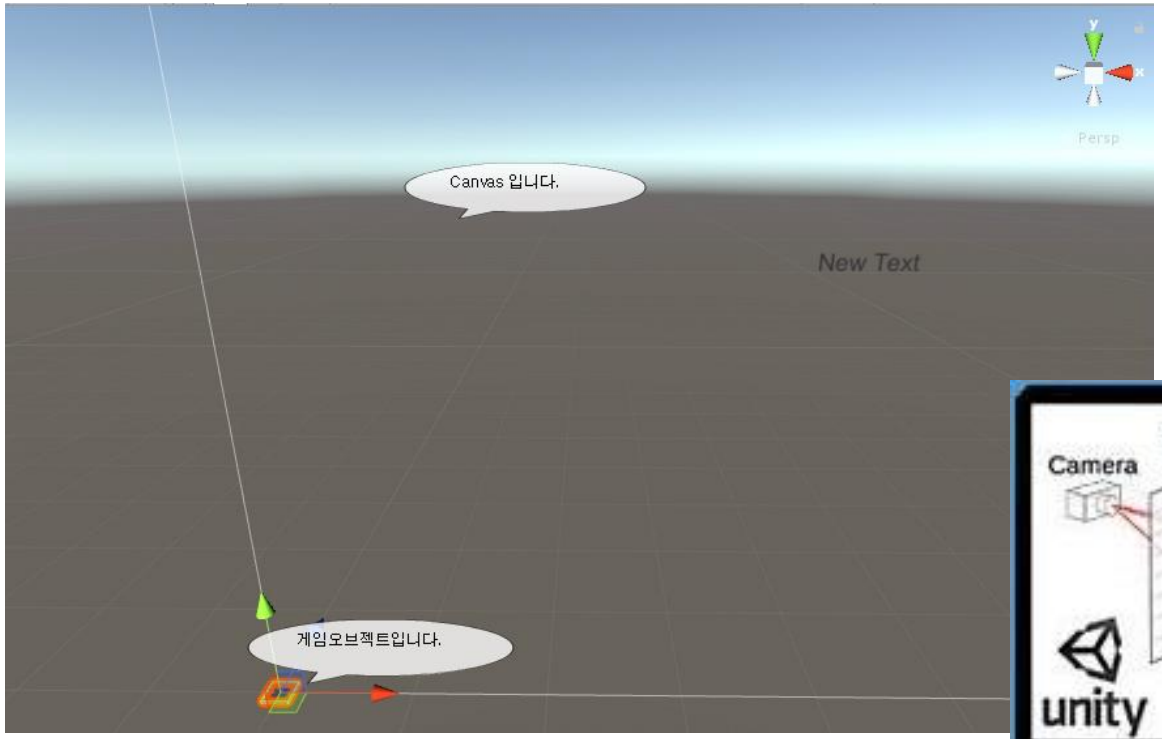
- 캔버스를 선택한 상태에서 메뉴 [GameObject – UI – Text]를 클릭하여 텍스트를 추가

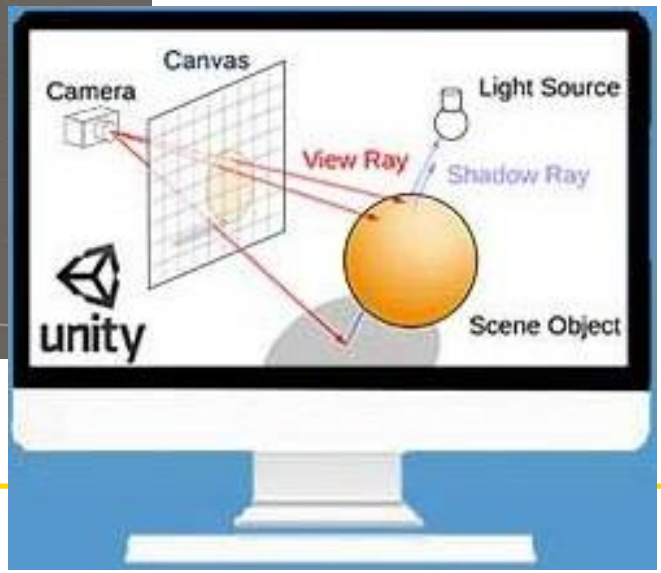


- 하이러리키 뷰에 Canvas의 자식 객체로 Text 객체가 생김
- UI가 화면에 표시되려면, 모든 UI 오브젝트는 Canvas의 자식으로 되어 있어야 함

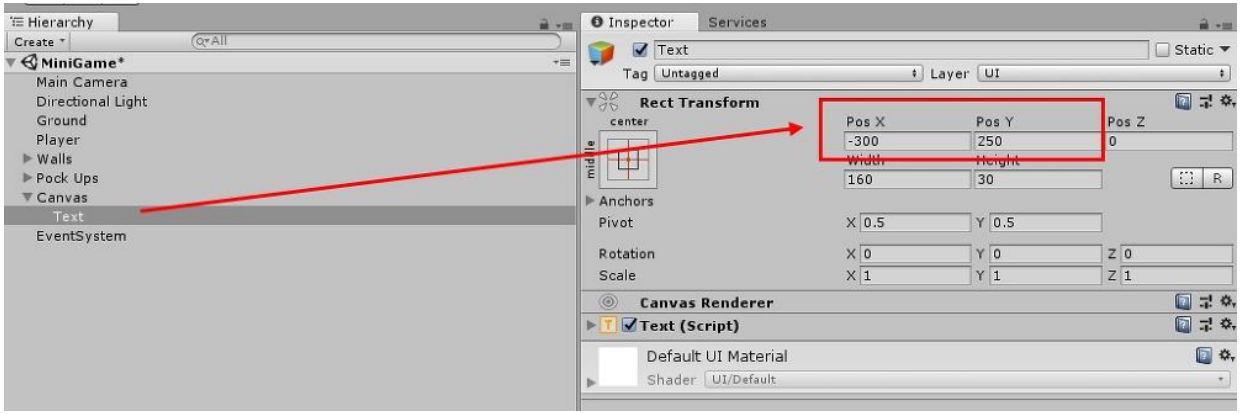


- Canvas를 씬 뷰에서 확인하면, 게임오브젝트와 거리상으로 떨어져 있는 곳에 배치됨
- 게임 뷰를 촬영하는 카메라 화면과 UI를 촬영하는 카메라 화면을 하나로 합쳐서 게임 화면에 출력됨

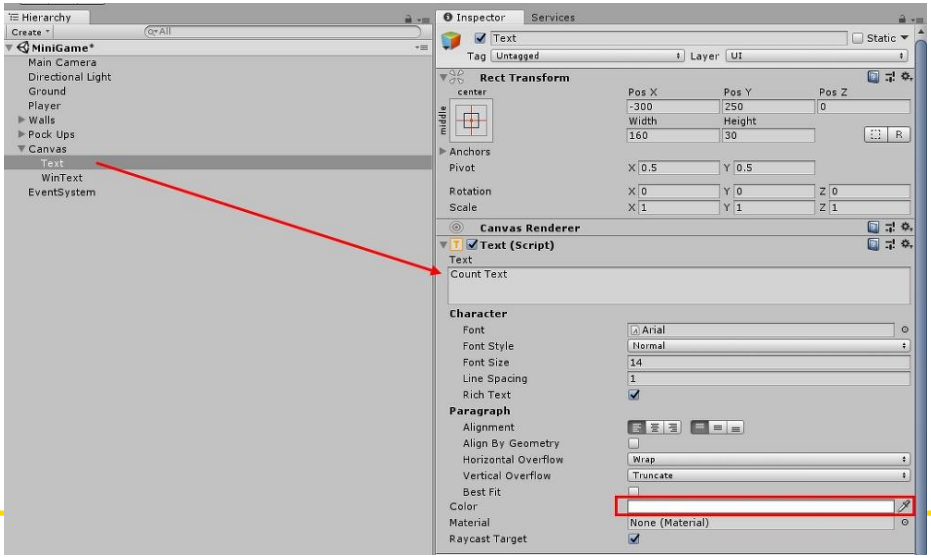




- Position의 X 값과 Y 값을 변경하여 좌측 상단에 표시되도록

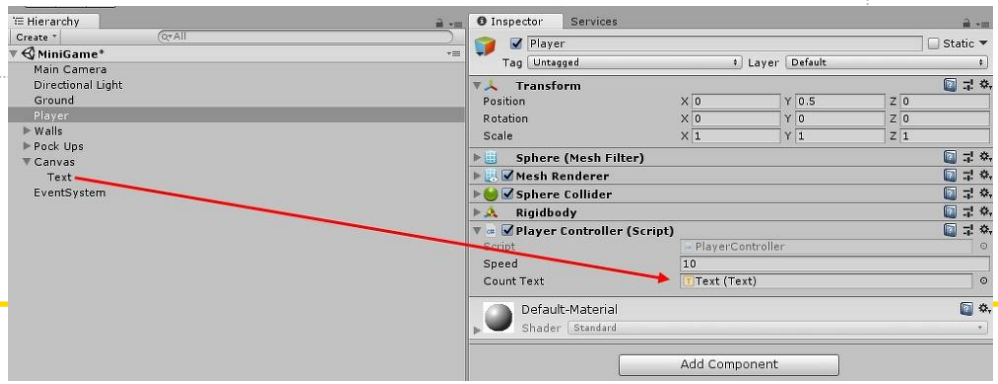


- Text 값을 입력한 다음, 속성을 수정



- 스크립트를 수정하고 public으로 선언한 변수를 인스팩터에서 연결

```
using UnityEngine.UI;
using TMPro;

public class CarFire : MonoBehaviour
{
    .....
    public Text countText;
    public TetMeshProGUI resourceText;

    void Start() {
        ....
        countText.text = “HP : ” + hp.ToString();
    }

    void OnTriggerEnter(Collider other) {
        if (other.tag == “Bullet”) {
            hp--;
            countText.text = “HP : ” + hp.ToString();
            resourceText.text = “Score : ” + hp.ToString();
        }
    }
}
```

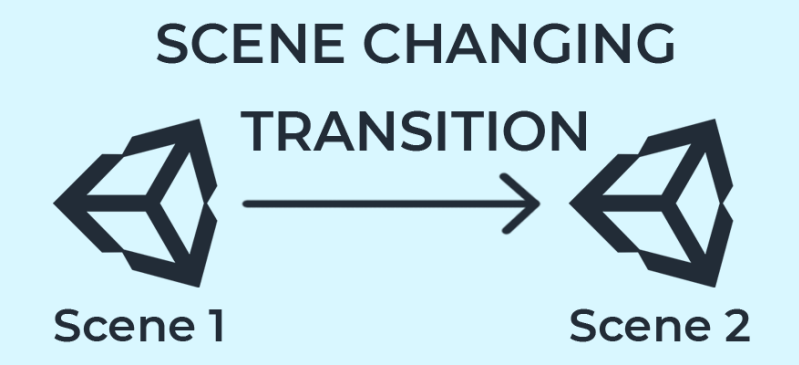


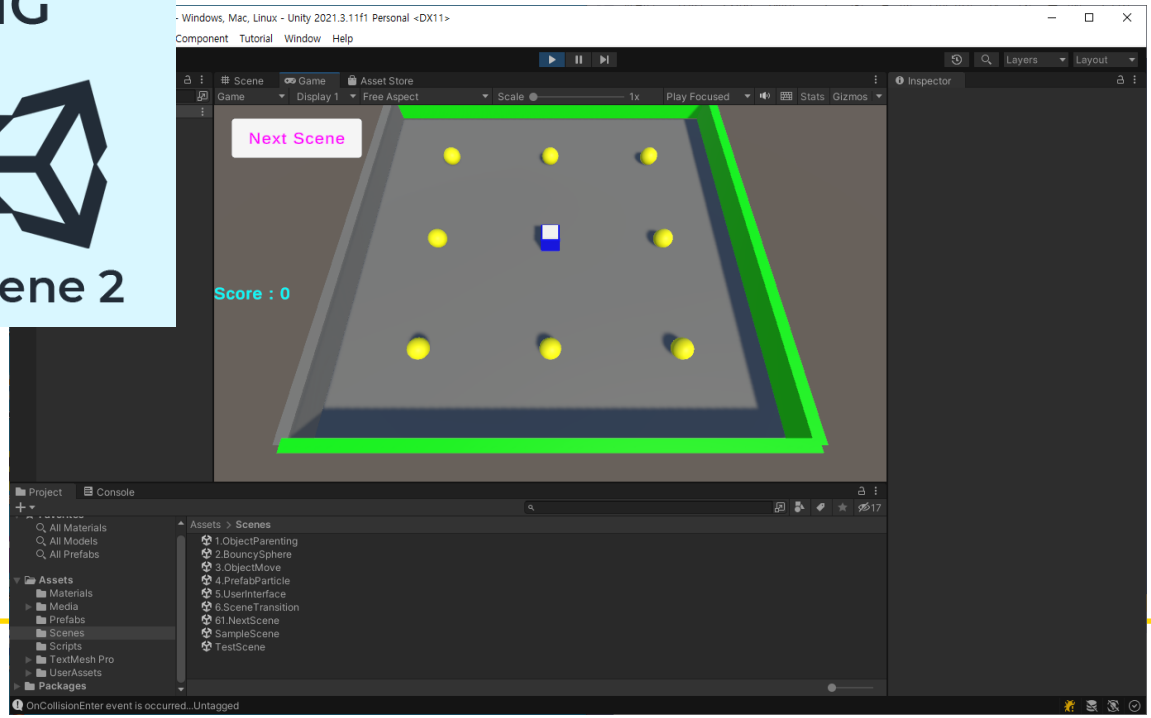
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TargetObjectCollision : MonoBehaviour {
    // 오브젝트 제거 파티클
    public Transform explosion;

    // AudioSource를 저장할 변수 선언
    AudioSource objectDestroySound;

    // Start is called before the first frame update
    void Start() {
        // 시작할 때 컴포넌트 열기
        objectDestroySound = GetComponent<AudioSource>();  // 씬이 로딩되면 AudioSource를 변수로 읽어 들임

    }

    // Update is called once per frame
    void Update() {}

    void DestroySelf(Vector3 pos) {
        //Debug.Log("DestroySelf method is called...");

        // 사운드(음향효과), 파티클, 점수 획득 등

        // 사운드(음향효과) -----------
        // AudioSource에 할당된 오디오 클립을 재생
        objectDestroySound.Play();

        // 파티클 -------------
        // 오브젝트 제거 시점에서 파티클 실행
        //Instantiate(explosion, pos, Quaternion.identity);
        // 파티클 오브젝트 생성 후, 삭제되지 않는다면 이렇게
        Transform parti = Instantiate(explosion, pos, Quaternion.identity);
        // Delay(지연)을 줘서 오브젝트 제거(destroy)
        Destroy(parti.gameObject, 2);

        // 객체 제거 --------------
        //Destroy(gameObject);
        // 오브젝트 제거 대신에, 오브젝트가 화면에 안보이게
        gameObject.SetActive(false);

        // 아이템 획득 수(score)를 올려줌
        GameManager.AddResource();
    }
}
```

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

public class GameManager : MonoBehaviour {
    [HideInInspector]
    public static int resource;

    //public Text score;
    // unity version이 올라감에 따라, UI text를 쓰기 보다는 TextMeshPro를 사용하는게 나을 듯
    public TextMeshProUGUI resourceText;

    // Start is called before the first frame update
    void Start() {
        // public으로 선언해서 (따로) GetComponent<>()를 할 필요 없음
        //resourceText = GetComponent<TextMeshProUGUI>();
        resource = 0;
    }

    // Update is called once per frame
    void Update() {
        //Debug.Log("Score : " + resource.ToString());
        resourceText.text = "Score : " + resource.ToString();
    }

    public int GetResource() {return resource;}
    public static void AddResource(int addCount) { resource += addCount; }
    public static void AddResource() { resource++; }
}
```

Scene Transition

- Scene 전환
  - SceneManager 를 위해, using UnityEngine.SceneManagement 필요
  - 전환하는 씬은 Building Setting 에 씬으로 등록되어 있어야 함
  - 씬 전환 시에, (필요한) 데이터 전달이 가능 (DontDestroyOnLoad() 사용)



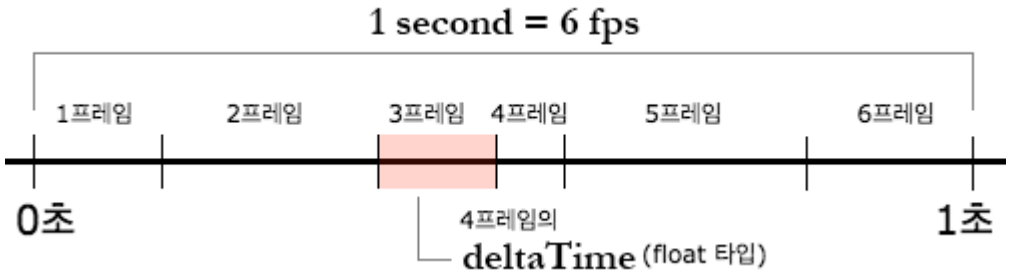


1. Text UI 오브젝트를 씬에 추가
2. GameManager 오브젝트를 생성하여 (게임 내에서) 전체를 다루는 스크립트를 작성
3. 특정 오브젝트와 충돌이 발생할 때 (아이템을 획득할 때) 점수를 부여하여 캔버스의 Text 에 표시
4. Button UI 오브젝트를 씬에 추가
5. 버튼을 클릭하여 다른 씬으로 전환

추가자료

Time.deltaTime

- The time in seconds it took to complete the last frame. Use this function to make your game frame rate independent (마지막(바로 전) 프레임을 완료하는데 걸리는 시간으로, 게임이 프레임 독립적으로 작동하기 위해 사용)
- 이전 프레임의 시작 시간부터 현재 프레임이 시작 시간까지의 간격 : deltaTime
- 만약 초당 6프레임인 상태에서, 현재 4프레임이라면, Time.deltaTime은 3프레임 (4프레임의 바로 전 프레임) 을 완료하는데 걸리는 시간



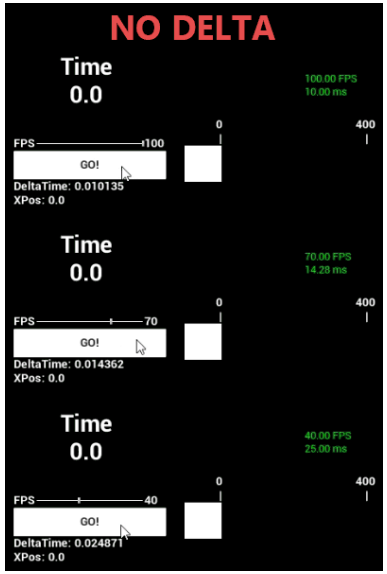
– Time.deltaTime 을 사용하는 이유

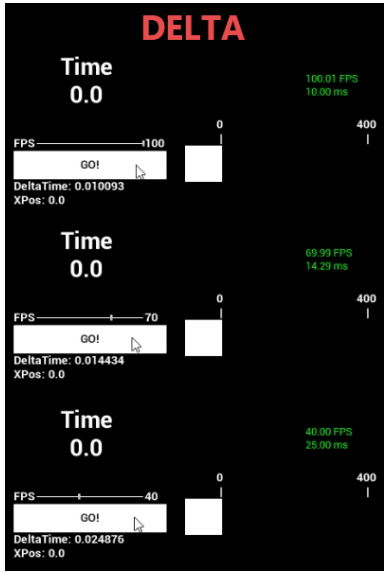
```
public class ExampleClass : MonoBehaviour {
    void Update() {
        //float translation = 10f;
        float translation = Time.deltaTime * 10;
        transform.Translate(0, 0, translation);
    }
}
```

- Update() 에서 Translate() 메소드를 통해 게임오브젝트를 Z 방향으로 10 유닛 만큼 이동
- translation 변수에는 deltaTime 만큼을 곱하고 있음
- 만약 deltaTime 을 곱하지 않으면 ?
  - 10 fps 로 동작한다면, 1초당 update()가 10번 호출되어, 100 유닛 만큼 이동
  - 60 fps 로 동작한다면, 초당 60번 호출되어, 600 유닛 만큼 이동
- deltaTime 을 곱하면 ?
  - 10 fps 에서, 1초당 update()가 10번 호출되면서 deltaTime (1/10)이 곱해져서, 10번 *(1/10)*10 유닛 만큼 이동
  - 60 fps 에서, 초당 60번이 호출되면서 deltaTime (1/60)이 곱해져서, 60번*(1/60)*10 유닛 만큼 이동
- FPS에 상관없이 동일한 단위의 이동을 지원하기 위해 (빠른 기기와 느린 기기 간에 동등한 조건을 지원)

- deltaTime 을 곱하므로, 동일한 유닛의 이동을 지원하도록

  - 늦은 FPS 에서는 (상대적으로) 큰 deltaTime 을 곱해주고, 빠른 FPS 에서는 (상대적으로) 작은 deltaTime을 곱해준다
  - 이를 통해 (FPS가 빠르건 느리건 상관없이) 동일한 이동 거리를 갈 수 있도록 조정





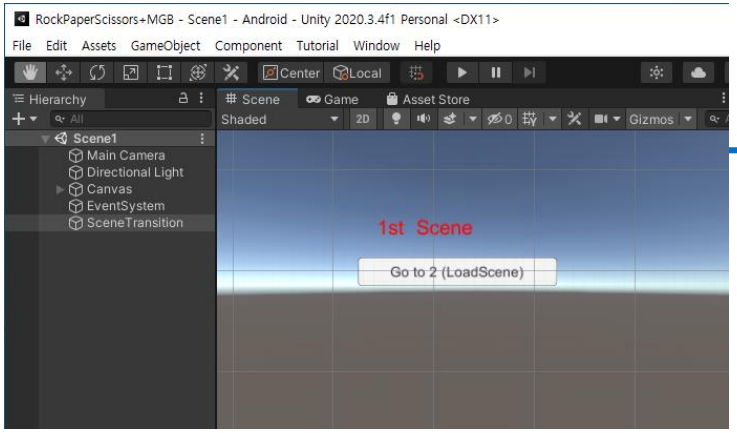
update() 안에서 deltaTime을 곱하여, 초당 정해진 유닛 만큼 이동하도록 지원하기 위함

deltaTime을 고정 값으로 사용하지 않는 이유는 프레임 간 처리 시간이 항상 일정하지 않기 때문임

- Example of Scene Transition
  1. Scene Transition using LoadScene()
  2. Scene Transition using LoadSceneAsync()
  3. Scene Transition with Parameter
  4. DontDestroyOnLoad() 로 설정된 게임 오브젝트 파괴

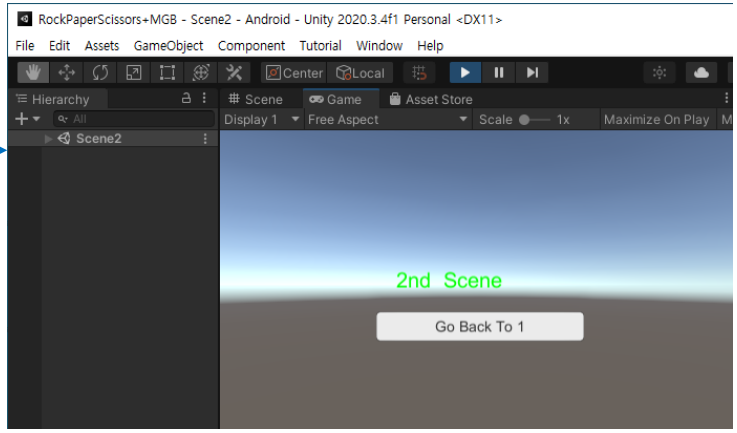
1 Scene Transition by LoadScene()

화면1

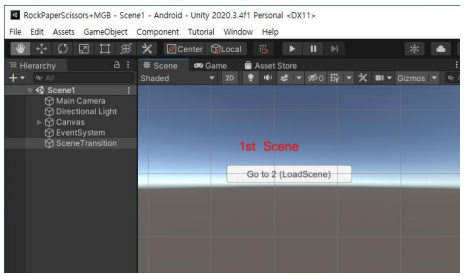


버튼을 클릭하여 씬 전환

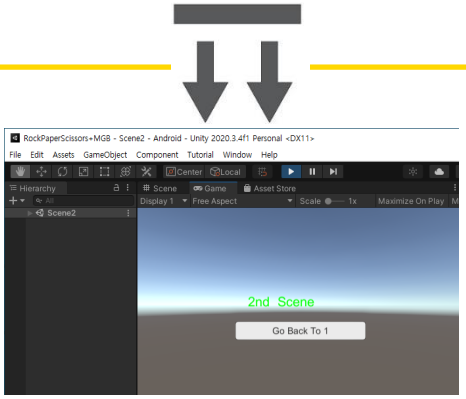
화면2



- [Go to 2] 버튼을 클릭하여 (현재의 Scene1에서 다음의) Scene2로 씬 전환



- Scene1 에서
  - 빈 오브젝트 (named SceneTransition) 생성
  - 스크립트 (named SceneTransitionContoller) 생성
  - SceneTransitionController 스크립트를 SceneTransition 오브젝트에 연결
  - 스크립트 수정 : 다음 슬라이드 참조
  - Canvas에 Text와 Button을 생성
  - Button의 OnClick() 함수에 SceneTransition 오브젝트를 연결하고, LoadScene()함수를 호출하는 함수를 연결



- Scene2 에서
  - 빈 오브젝트 생성 (SceneTransition)
  - 스크립트 SceneTransitionController를 오브젝트 SceneTransition에 연결
  - Canvas에 Text와 Button을 생성
  - Button의 OnClick() 함수에 SceneTransition 오브젝트를 연결하고 LoadScene()함수를 호출하는 함수를 연결

```
using UnityEngine.SceneManagement;

public class SceneTransitionController : MonoBehaviour
{
    .....

    public void CallScene1() {
        SceneManager.LoadScene("Scene1");
    }

    public void CallScene2() {
        SceneManager.LoadScene("Scene2");
    }
}
```

2 Scene Transition by LoadSceneAsync()

- 비동기적으로 Scene을 호출
  - 유니티는 씬을 불러올 때, 동기적으로 호출하는 LoadScene() 대신 LoadSceneAsync()를 사용할 것을 권장
  - LoadScene()은 중지 또는 약간의 지연 등이 발생할 수 있기 때문
  - LoadSceneAsync()은 AsyncOperation 클래스를 리턴 값으로 전달
  - AsyncOperation 프로퍼티로 현재 씬의 호출 상태를 알 수도 있음

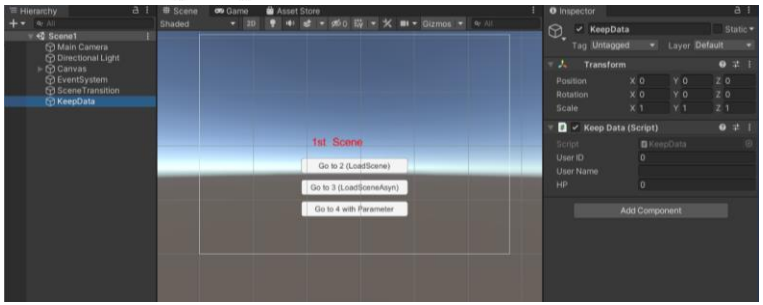
- Unity Documents 에서 발췌

  "In most cases, to avoid pauses or performance hiccups while loading, you should use the asynchronous version of this command which is: LoadSceneAsync()."

```
using UnityEngine.SceneManagement;

public class SceneTransitionController : MonoBehaviour
{
    .....
    void Update() {
        // Space bar 키 입력
        if(Input.GetKeyDown(KeyCode.Space)) {
            LoadNextScene();
        }
    }

    public void CallScene1() {
        SceneManager.LoadScene(“Scene1”);
    }

    public void CallScene2() {
        SceneManager.LoadScene(“Scene2”);
    }

    public void LoadNextScene()
    {
        // 비동기적으로 Scene를 불러오기 위해 Coroutine()을 사용
        StartCoroutine(LoadMyAsyncScene());
    }

    IEnumerator LoadMyAsyncScene()
    {
        //Debug.Log("LoadMyAsyncScene() is called");
        // AsyncOperation을 통해 SceneLoad() 정도를 알 수 있음
        AsyncOperation asyncLoad = SceneManager.LoadSceneAsync("Scene3");

        // Scene를 불러오는 것이 완료되면, AsyncOperation은 isDone상태가 됨
        while (!asyncLoad.isDone)
        {
            yield return null;
        }
    }
}
```

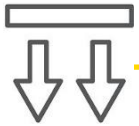
3 Scene Transition with Parameter

- 여러 개의 Scene으로 구성된 게임은 Scene들 간의 이동 시에 데이터 공유 등이 필요할 수 있다. 예를 들어, 캐릭터가 A 씬에서 B씬으로 이동할 때, 캐릭터의 HP, MP 등의 데이터를 넘겨줘야 한다
- 실제 게임 오브젝트를 넘기는 것이 아닌 데이터를 파일로 저장하고, 새로운 씬이 호출되면 저장된 값을 불러와 사용하도록 설정

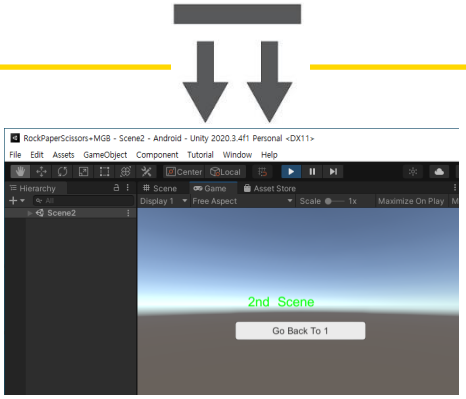


- Scene1 에서
  - 씬 전환 시에 넘겨주는 데이터를 게임 오브젝트로 보존
  - `Awake()` 에서 `DontDestroyOnLoad()` 함수를 통해, 전환 시점에서 게임 오브젝트의 폐기를 방지

    (씬이 바뀌어도 삭제되지 않은 객체의 변수에 접근이 가능함)
  - 데이터 타입은 public 로 선언



```
public class KeepData : MonoBehaviour
{
    public string username;
    public int HP;

    void Start() {
        username = “YongHwan”;
        HP = 100;
    }
    .....

    private void Awake() {
        DontDestroyOnLoad(this.gameObject);
    }
}
```



- Scene4 에서
  - 삭제되지 않고 넘어온 게임 오브젝트를 탐색
  - 탐색한 게임 오브젝트에 있는 (연결된) 스크립트를 불러옴 (GetComponent)
  - 스크립트에 있는 변수 사용

    (해당 변수에 있는 값을 자유로이 사용이 가능함)

```
using UnityEngine.UI;

public class GetData : MonoBehaviour
{
    // 매개변수로 넘어오는 값을 받기 위한 임시 오브젝트 생성
    GameObject paramObj = null;
    Text UItextName;
    Text UItextHP;

    void Start() {
        paramObj = GameObject.Find("KeepData");
        if (paramObj != null) {
            string name = paramObj.GetComponent<KeepData>().username;
            int HP = paramObj.GetComponent<KeepData>().HP;

            UItextName.text += name;
            UItextHP.text += HP.ToString();
        } else {
            Debug.LogWarning("[경고] 매개변수가 넘어오지 않았습니다");
        }
    }

    ...
}
```

4 DontDestoryOnLoad() 게임 오브젝트 파괴하기

- DontDestoryOnLoad()로 설정된 게임 오브젝트도 필요에 따라 파괴
  - Can't 가 아니라 Don't 임 (파괴할 수 없는 것이 아니라 파괴하지 않는 것임)
- Destory(gameObject); 로 파괴할 수 있음

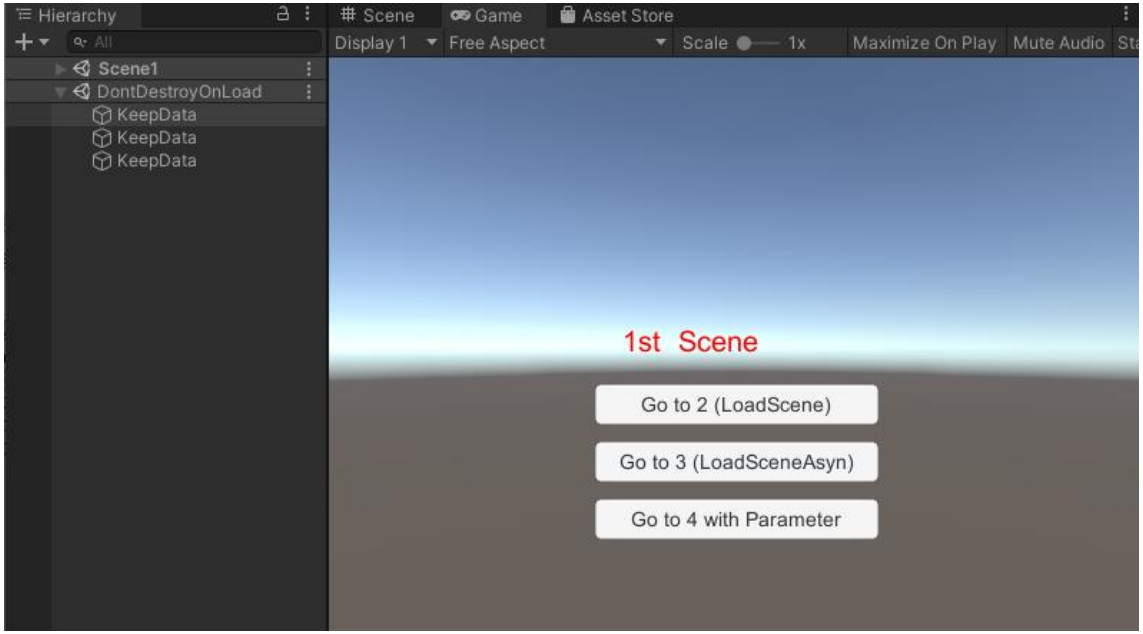
– 설계할 때 유의 사항

- DontDestoryOnLoad() 가 적용된 게임 오브젝트가 있는 씬과 다른 씬을 여러 차례 왔다 갔다 하는 경우

(예) TestScene에 진입하면, 파괴하지 않은 게임 오브젝트가 DontDestoryOnLoad 영역으로 이동되어 씬을 이동해도 파괴되지 않음. 그 다음 OtherScene으로 이동했다가 다시 TestScene으로 돌아오면 어떻게 되는지 확인한다

- 씬이 새로 불러올 때는 해당 씬의 초기 상태로 불러오기 때문에, 기존에 파괴되지 않은 게임 오브젝트와 새로이 생성된 게임 오브젝트가 중복적으로 나타남
- 즉, 두 씬을 왕복하면 같은 게임 오브젝트가 계속적으로 누적되는 문제가 발생

  불필요한 메모리 사용을 증가시키며, (중복된 게임 오브젝트로 인해) 예측할 수 없는 오동작이 발생할 수도 있음



4 Scene Transition with Parameter (using Prefab)

- 씬을 전환하면서 데이터를 넘겨줘야 하는 경우,

Tank Game

화면 흐름 예