20230524_구조설명회

고민했던 내용의 본질을 잊지 않기위해 작성했다.
그리고 결론을 내려, 앞으로 작성할 코드에 타당성을 적어보고,
추후 구현된 인스턴스, 클래스를 사용할때 사용될때 이해를 도모하는 글이 될듯.

🏛️ 1. 요약

1. Entity의 정의 그리고 인터페이스 설명회 그리고 Projectile과 관계

2. 커서와 키보드를 인식하는 클래스 그리고 Entity의 관계

3. Projectile, Modifiyer, Equipment의 구현 & 데이터 교환법

🏛️ 1. Entity에 대한 정의

🤔 1. Entity 특징

① Entity Component

- EntityData를 포함(Composite)하는 컴포넌트
- Entity라면 할 수 있는 가장 기초 임플란트의 로직(*IEntityAddressable*)또한 구현되어 있다.

② IEntityAddressable

- 최소 Entity라면 적어도 할수 있는 행동을 구현하는 Interface

③ Collider Component

- Entity라면 꼭 가지고 있는 컴포넌트가 된다.
- 콜라이더를 가졌다, RayCast, OnTrigger이 가능하다는것
- 로직, 데이터 변경이 가능하다.

④ WorldSpace에 존재할 수 있다

- 예시 Player, Enemy, 보스 몬스터, 함정

⑤ 효과에 대해 Affectable 하다 (ModifierHandleing)

- 디버프 상태, 버프 상태, 시너지 상태를 가질때
- 플레이어 내에 정의 되지 않는 로직(Modifiyer)이 주입되어 실행 가능해진다.

🤔 2. Entity와 Projectile의 본질

▶ 공통점

1. **Projectile** 또한 *Collider Component*를 가지고 있으며
2. *World Space*에 존재 하며
3. EntityData을 사용한다는점에서 Entity랑 공통점을 가지는것 같다.
4. 심지어 상황에 따라서는 *Mouse & Key Interactable* 이 가능하다는것이다.

▶ **너무나 비슷해 보이는 이 두녀석의 차이점은 과연 뭘까?**

- **Entity** : `Entity - Entity` 끼리는 EntityData를 **"직접 Set"할 수 없다는것이다**. GetDamage, Attack, EntityData 변환과 같은 **"로직, 데이터접근"** 있어서 꼭 **매개체가 필요하다.**
  - Entity는 꼭 무조건 Projectile을 소환할 능력이 있다.

```
매개체의 예시는 다음과 같다
1. Entity - Projectile - Entity
2. Entity - Mouse & Key Input's - Entity
```

- **Projectile** : Entity의 매개체가 되어서 **EntityData를 접근해 "Get하여 Set"** 하는 차이점이 있다. 즉, `Entity - Projectile - Entity` 관계가 된다.
  - Projectile은 **Entity 로직(*IEntityAccessable*)에 접근할 수 있다**는게 결정적 차이다.
  - 심지어 Entity의 매개체가 될뿐만 아니라 *Collider Component*를 가지는 모든 GameObject의 매개체가 될 여지가 있다

```
총알           : IEntityAccessable.GetDamage(_DamageAmount);
땅에 떨어진 아이템, 장비 Gear : EntiyData.Gear;
```

▶ **코드 구현에 있어서 매우 간단하게 분류하면**

- Entity는 IEntityAccessable을 구현하며, EntityData를 포함(Composite) 한다
- Projectile은 아무것도 상속 안한다. 아이템이나 룰렛이라면 Entity처럼 Interactable만 구현된다는 것이다.

◉ 따라서 Entity & Projectile은 완벽히 같은 부모를, 혹은 ,같은 인터페이스를 구현하지 않는다. (진짜 많이 비슷해보여도 본질이 다르다를 이해하면 좋을것이다.)

🤔 3. 여기서 질문

변수 이름 짓기다 : 적어도 이름만 봐서 "아! 어떠어떠한거 가지고 있겠고, 이러이러한거 하겠구나" 라는게 명확이 떠올려 질 수 있어야 한다.

1. Entity 이 단어가 위의 역할 설명을 잘 해주나?
   - YES : 변수 이름을 바꿀 필요가 없음
   - No : 제일 잘 설명하는 변수 이름을 지어보자
2. Projectile 이 단어가 위의 역할 설명을 제대로 하는것 같나?
   - YES : 변수 이름을 바꿀 필요가 없음
   - No : 제일 잘 설명하는 변수 이름을 지어보자

🏛️ 2. 커서와 키보드를 인식하는 클래스

🤔 1. 다음과 같은 예시에서 사용된다.

```
1. "떨어진 장비"에 커서를 올려놓으면 UI가 뜬다
    를 구현하기 위해서는
        > "떨어진 장비"는 커서를 인식을 한다.
        > "떨어진 장비"에 커서가 가까이 가야 된다..
        > "떨어진 장비는" 커서가 가까이 왔다는걸 인식하고 나서 UI가 뜨는 반응이 생긴다.

2. "아이템"을 줍기위해 플레이어가 일정 범위에 다가가니 상호작용키가 활성화 되었다.
    를 구현하기 위해서는
        > "아이템"은 플레이어가 어느정도 가까워 졌다는것을 인식해야 한다.
        > 플레이어가 충분히 가까이 왔다는걸 인식해 인풋이 가능하다는걸 UI가 반응한다.

3. "몬스터" 에 커서를 가까이 대면 "에임 주목"을 한다.
    를 구현하기 위해서는
        > "몬스터"는 커서 인식을 한다.
        > "몬스터"에게 커서가 가까이 가야된다.
        > "몬스터"는 커서가 가까이 왔다는것을 인식하고 UI가 뜨고, 마우스는 달라붙어야 하고, 화면은 그에 맞춰 시야가 가게 된다.

4. UI는 마우스 커서를 인식하는것 뿐만 아니라 클릭, 주목, 드래그, 클릭 떼기
    심지어 키보드 인풋도 받을수가 있어 상하좌우에 따라 주목이 되는 반응을 할 수 있다.
    를 구현하기 위해서는
        > "UI"는 마우스 동작을 인식한다.
        > 키보드 동작또한 인식한다.
```

🤔 2. Interactable 인터페이스를 만들기전에 고민할것.

부류 A : 1, 2, 3

- Entity : 레벨(맵)과 생명주기를 같이하며, *Collider Component*을 가지고 있다.
- 마우스와 키보드를 인식하여, 그것으로 로직을 조작할 수 있다. *Interactable*을 구현한다.

부류 B : 4

- 마우스와 키보드를 인식하여, 그것으로 로직을 조작할 수 있다. *Interactable*을 구현한다.

부류 B와 부류 A가 두 부류가 본질적으로 같은 집합에 포함관계인가?
다르게 말하면 예시 1,2,3이 할수 있는 일이 완전히 예시 4 또한 할수 있냐?

- 공통점
  - 마우스 커서를 인식하고, 키보드 인풋을 받는다. *Interactable*
- 차이점
  1. 부류 A는 *Collider Component*가 있으며 맵에 존재한다.
  2. 부류 B는 그저 마우스와 키보드를 인식하며, WorldSpace에 존재할수 없다. *Collider Compoent가 없다는것*

그저 마우스, 키보드에 대한 행동을 정의하고 있다고해서 모두 *Interactable*로 일반화(그룹 지을수) 하기는 어렵다. 일부 Interactable는 Collider을 안가지며, 맵과 무조건적으로 생명주기를 의존하지는 않기때문이다. **Ex). UI**

◉ 따라서 두 부류는 완벽히 같은 부모를, 혹은 ,같은 인터페이스를 구현하지 않는다

🤔 3. Entity + Interactable = ?

따라서 부류 A를 위한 인터페이스를 정의해 UI와 유기적인 Entity를 만들어야 한다.
맵에 생명주기 생명주기를 의존하며. 마우스, 키보드를 인식할 수 있는것을
다음을 명시해야 한다. Entity라 해서 Interactable이 가능한것은 아니다.

- 특징 :
    1. Collider Component
        1. OnTrigger :

            ```
            서로간의 로직에 접근이 가능하다.  Ex) Projectile
            Keyboard || MouseInput Check  Ex) 특정 범위에 들어가야 반응
            ```

        2. RayCast's Target -> Mouse Input Check
    2. Entity Component
        1. 데이터를 Get, Set이 가능한 여지가 있다.

            ```
            Projectile을 통해 로직, 데이터 개입이 가능하다,

            Get : Projectile : Entiy의 데이터 퍼센트 비율로 참조하기 위함
            Set : 버프, 디버프, Equipment, Skill, Gear 데이터 접근
            ```

    3. Interactable
        1. 마우스 인풋, 키보드 인풋에대한 로직을 가진다.

🤔 4. 여기서 질문

변수 이름 짓기다 : 적어도 이름만 봐서 "아! 어떠어떠한거 가지고 있겠고, 이러이러한거 하겠구나" 라는게 명확이 떠올려 질 수 있어야 한다.

🏛️ 3. Projectile, Modifiyer, Equipment의 구현 & 데이터 교환법

🤔 1. Projectile의 구현

- 다음이 만족해야한다.
    1. Entiy의 데이터와 로직에 접근이 가능해야한다. ✅
    2. Entity & Projectie에 의해 Instantiate되야 한다. ⚠️
        - 룰렛(Projectile) 또한 Item을 내뿜는다.
    3. OnCollision, OnTrigger이 구현되어 있다. ✅
    4. 즉시 삭제가 되든, 자가 삭제가 되든.. 가능하다. ✅
    5. 나를 생성한 OwnerEntity가 있으며, 간섭 타겟인 TargetEntity또한 알고 있다. ✅
        - 플레이어의 공격 이펙트가 플레이어를 때려서는 안된다

6. Attack하는 타입인지, Modifiy하는 타입인지 알수 있다. ✅
7. 나를 생성한 OwnerEntity의 데이터를 참조해 전달 가능하기도 한데 Projectile자기 자신만이 가지고 있는 데이터또한 전달 가능하다 ⚠️
    - 독데미지를 가지고 있는 플레이어 상태와 상관없이 Projectile
    - 밟으면 불데미지 입는 함정
    - Equipment 데이터를 가지고 있는
    - Gear + 100 올라가는 금화

- 코드 구현

```
// 이로서 어펙터블은 스크립터블로 관리하는게 좋을지도 모르겠다라는 생각이 든다.
// 어펙터블은 디버프, 버프, 시너지가 있으며. 이것들을 조금 수정하는 일이 생길것이다.
// 그렇게 된다면 장비또한 완전한 데이터 기반으로 정의 가능하다.
// 그말은 스크립터블 오브젝트로 갈아끼우면서 틀을 정의 가능하다.
//
// 기존의 문제를 다음오르 해결할 수 있다.
// 1. 이큅먼트의 구현을 코드적 정의가 아니게 할 수 있다.데이터 기반이 가능하다 데이터를 갈아 끼울 수 있다.
// 2. 투사체또한 데이터 기반이 가능하다. : 투사체 또한 스크립터블을 갈아끼우면서 복제 가능하다.
//
// 그걸 가정하고 구현해보자

public enum E_ProjectileType {
    harm, nutral
}

public class Projectile : Monobehaviour, IDestroyAccessable{
    public E_ProjectileType projectileType;
    public bool isMove; public float moveSpeed;
    public Modulator modulator;

    private float damageAmount;
    private IEntityAddressable owner        ;
    //List<UnityAction> destroyCallback();
    //List<UnityAction> projectileCallback();
    public void Initialize(owner) {
    }
 ㄴ
    private void OnTrigger(Collision other){
        IEntityAddressable targetEntity = other.GetComponent<   IEntityAddressable>();
        if(projectileType == E_ProjectileType.harm) {
            target.GetDamage(_amount);
        }
        if(projectileType == E_ProjectileType.nutral) {
            target.
        }
    }
    private void Update(){
        if(isMove) {Rigid.velocity = 1;}
    }
    public Destroy(float time) {
        destroyCallback?.Invoke();
        Destrot(gameobject, time);
    }
}
```