YourFurnace

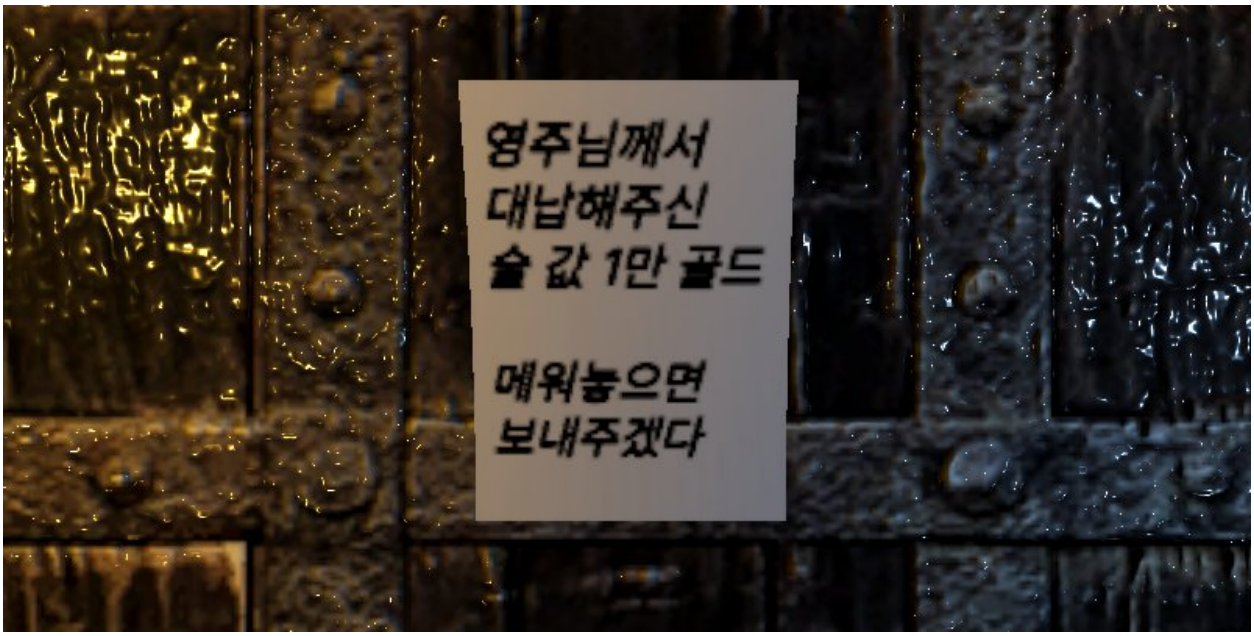
난수 생성 대장간과 장비 강화 시뮬레이터

작업물 세부 안내

한승우 작성

목차

프로젝트 개요



시작하면 보게 될 화면

본 프로젝트는 순전히 **개인적인 흥미에서 시작**되었다. 잦은 과제로 누적되는 피로, 스스로의 코드를 리뷰할 시간 없이 반복적으로 다음 과제를 진행함에 따라 오는 **권태감은** 누가보아도 향후의 습작 활동에 **방해가 될 것**이라 보기 충분했다. 하여, 이번 프로젝트로 쳐진 분위기를 반전시키고자 하였고, 인터넷 방송 등지에서 자주 볼 수 있는 **'아이템 강화' 를 테마로** 하여 우스꽝스러운, 혹은 **어처구니 없는 상황을 유발해 웃음을 지어보고자** 하였다.

본 프로젝트는 Unity 6000.0 을 사용하였으나, XR Controller 의 중력 제공 컴포넌트Component를 가져다 쓰기 위해 XR 프레임워크와 Unity 버전을 한 차례 업데이트 하였고, 최종적으로 **Unity 6000.2** 프로젝트가 되었다. 세부 설명서를 읽고 있거나 Github에서 프로젝트를 클론Clone해- 에셋이 없으니 에디터가 제대로 보여주지 못하겠지만 -직접 확인해보고자 하는 독자들은 이 부분을 유의하면 좋을 것 같다.

해당 세부 설명서가 서술하는 프로젝트의 스크립트는 Github 에서 확인할 수 있다. 디지털 문서로 열람중인 독자를 위해 링크를 아래에 남긴다. 이 문서도 해당 저장소에서 찾을 수 있다:

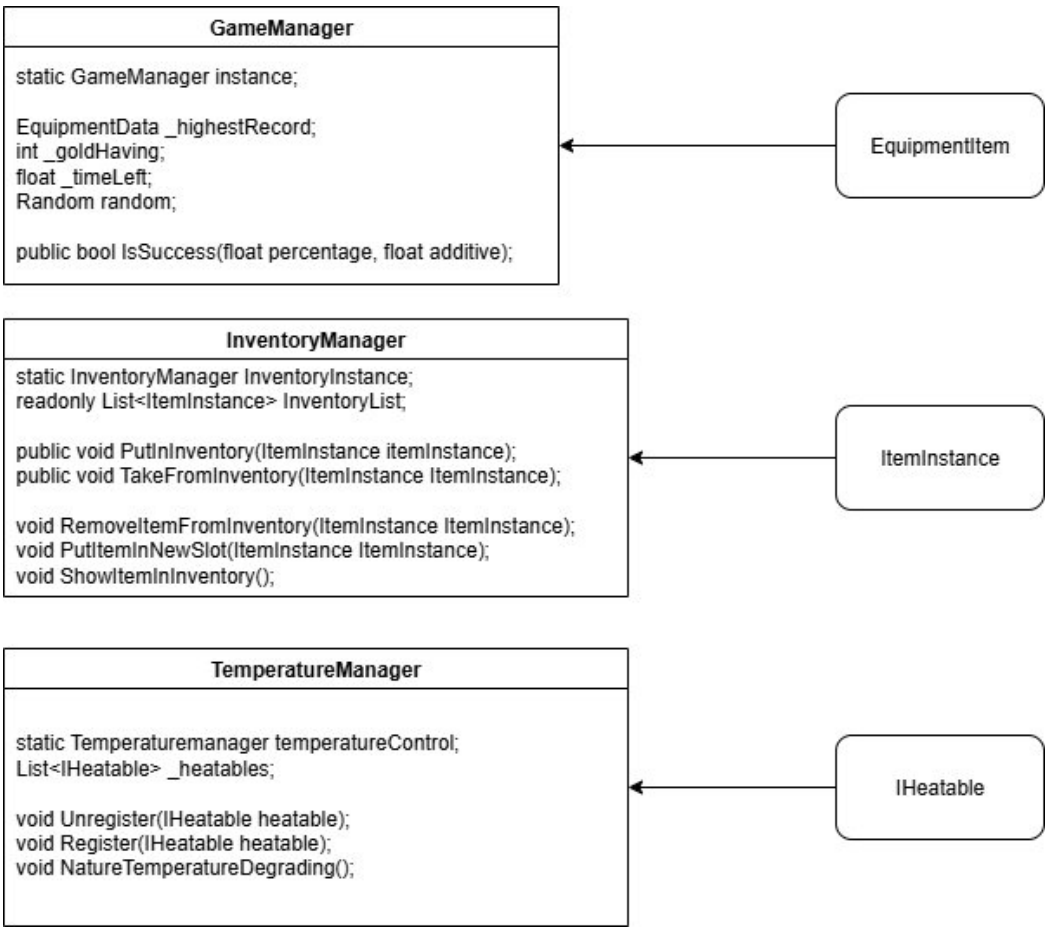
https://github.com/Yuny2036/YourFurnace

공통

게임은 실내와 실외 두 개의 씬Scene으로 나뉜다. 사실, 실외는 Unity의 Terrain Tool을 한번 구경해 보고 싶은 마음이 있어서 핑계 삼아 추가하였으며, 강화 확률에 보정값을 주는 **버섯**과 문을 열면 실내로 넘어가는 **Scene-switching** 을 제외하면 딱히 볼 거리 없는 작은 땅덩어리와 Climbable한 나무판 뿐이다.

두 씬에서 공통적으로 존재하는 오브젝트들은 으레 그렇듯 **전역적인 데이터들을 관제**하는 **관리자 Manager** 오브젝트들이다. 게임에는 다음의 관리자들이 있다:

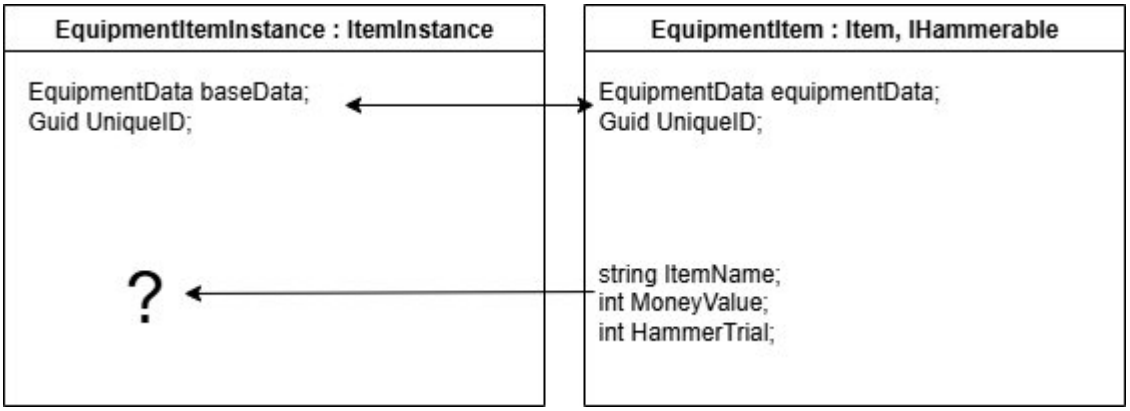
- 남은 시간, 골드 보유량과 최고 강화기록을 **기록**하는 게임관리자GameManager
- **인벤토리** 이벤트를 처리해줄 인벤토리관리자InventoryManager
- 데워질 수 있는IHeatable 오브젝트들을 구독해 **온도**를 내리는 온도관리자TemperatureManager



관리자 오브젝트가 여럿 있지만 막상 해보니 관리자라고 일이 많은건 아니었다

인벤토리

누군가 만약 *'구현했던 기능들 중* ***생각과 현실이 달랐던 것****을 하나 뽑으라'* 고 하면, 난 인벤토리를 뽑을 것이다. 그저 막연히 '그거 기존 인스턴스들 정보 json 파일로 정리해서 저장하고 나중에 부르면 되는거 아니야?' 라고 생각했었지만, 지금의 내가 그 모습을 볼 수 있다면 주저없이 '그거 아니야!' 라고 답했을 것이다.



장비의 경우는 단순히 인벤토리 정보가 덜 필요했지만, 소모품은 '최대 보유량' 이라는 개념이 인벤토리 정보에 ***만*** *있다.*

깨달은 개념 중 가장 핵심적인 것은 **월드 인스턴스 ≠ 인벤토리 인스턴스** 라는 것이다. 단순히 월드에서 보는 오브젝트의 데이터가 인벤토리에서 모두 필요한 것이 아닌데다, **논리적인 이유**로 인벤토리에서 쓸 별도의 데이터도 요구받는다.

여기서 하나 분명히 해야 할 것은 '**논리적인 이유**' 부분인데, 본인이 어떤 인벤토리를 구현하느냐에 따라 로직의 구조도 다채롭게 바뀌기 때문이다. 분량을 줄이고자 내 인벤토리로 넘어가자면:

1. 인벤토리는 List<ItemInstance> 타입으로 **4개의 아이템**(= 슬롯 4개)만 받아야 한다.
2. 장비 아이템은 **종류불문** 아이템 **슬롯 1개**를 차지한다.
3. 소모품 아이템은 아이템마다 최대 스택 범위가 있어, **범위 내에서는** 아이템 **슬롯 1개**의 취급이다.

정도로 볼 수 있다. **마인크래프트**의 인벤토리 시스템이 이와 유사하다.

```
public void PutInInventory(ItemInstance itemInstance)
   {
       switch (itemInstance)
       {
           case EquipmentItemInstance eii:
               // Is the exactly same item already in the inventory?
               if (InventoryList.Contains(eii)) throw new ArgumentException("You're
trying to put the exact same entity in. How did you do..?");
               PutItemInNewSlot(eii);


               break
```

인벤토리 넣기 장비 아이템 로직

장비 아이템은 구현이 *어렵지 않다*. 인벤토리 시스템에 List 타입을 사용했으므로, List.Contains(instance)를 사용해 **동일 아이템 검수가 가능**하고, 필요하다면 Guid 비교로 깐깐하게 비교하는 로직을 짤 수도 있다. 이후엔 List.Add(instance) 로 과정이 끝나게 된다.

정말 머리 아픈 부분은 소모품에 있었다. 위의 3. 에서는 한 줄로 간단히 서술했으나, 자연스럽고 논리적으로 오류없는 인벤토리를 구현하려면 아래의 항목이 요구된다:

a. 넣으려는 종류의 소모품이 **인벤토리에 존재하는지** 확인한다.
b. 넣으려는 종류의 소모품이 있다면, 인벤토리 속 **소모품의 남은 공간이 넉넉한지** 아닌지 확인한다.
c. b. 에서 공간이 넉넉하지 못했다면, 다른 슬롯에 넣는데, 이 때 **비어있는 슬롯이 있는지** 확인한다.

'가방에 넣는다' 라는 기능을 별도의 메서드로 구현해도 3개의 로직이 필요하다. 여기까지 오면 마인크래프트 스타일의 기본적인 인벤토리 기능이 구현된다.

```
case PropsItemInstance pii:
                // Get the existing item from the inventory.
                var existingItem = InventoryList
                .OfType<PropsItemInstance>()
                .FirstOrDefault(item => item.baseData.baseID == pii.baseData.baseID);

                // When there's the item.
                if (existingItem != null)
                {
                    // Calculate how many stack can be added into the existing one.
                    int stackLeft = existingItem.baseData.MaximumStacks -
existingItem.CurrentStacks;

                    // Space left is bigger than one you're trying?
                    // If yes,
                    if (pii.CurrentStacks <= stackLeft)
                    {
                        existingItem.CurrentStacks += pii.CurrentStacks;
                    }
                    // If no,
                    else
                    {
                        existingItem.CurrentStacks += stackLeft;

                        // Get remaining items' count.
                        var remainingItem = new PropsItemInstance(pii.baseData,
pii.CurrentStacks - stackLeft);

                        // Put em in the new slot
                        PutItemInNewSlot(remainingItem);
                    }
                }
                else
                {
                    PutItemInNewSlot(pii);
                }
                break;
```

인벤토리 넣기 소모품 아이템 로직

인벤토리에 아이템을 넣을 때와 인벤토리에서 아이템을 뺄 때의 **이벤트에 쓸 기능**을 **먼저 개발**하고, **List를 직접 다룰 메서드는** 내부 메서드private 로 **접근 제한**을 두었다. 공개 메서드에서 여러번 중복이 되기도 하고, 노출되는 메서드에 필드 값을 조작하는 코드를 줄이고자 함도 있었다.

```
private void PutItemInNewSlot(ItemInstance itemInstance)
{
    if (InventoryList.Count >= 5) throw new ArgumentOutOfRangeException("Inventory is
full; Maximum is 4.");
    InventoryList.Add(itemInstance);
}
```

List를 직접 다루지 않고 따로 메서드를 이용, 하는 김에 겸사겸사 슬롯체크

인벤토리와 상호작용할 때 **switch-case 문으로 아이템의 타입**을 구분지었다. C#의 switch-case는 비교할 대상의 타입이 **case의 타입인지 확인**하고 넘길 수 있다. 객체-지향 프로그래밍OOP의 핵심 중 하나인 **다형성**이 잘 살아나는 문법이라 생각한다.

코멘트

1. 프로젝트 구상에는 있었지만, 습작이 장기화되는 것을 지양하였기에 일부 기능이 아직 미구현 상태로 남아있다. 대표적인 것이 게임관리자인데, 현재 게임에 점수나 골드 보유량은 필드Field만 적혀있을 뿐, 실제로 기능하지 않는다. 다만, 난수 생성에 쓸 Random 인스턴스를 게임관리자가 들고 있고 해당 인스턴스에서 무작위 확률을 추출하는 역할 만을 할 뿐이다.

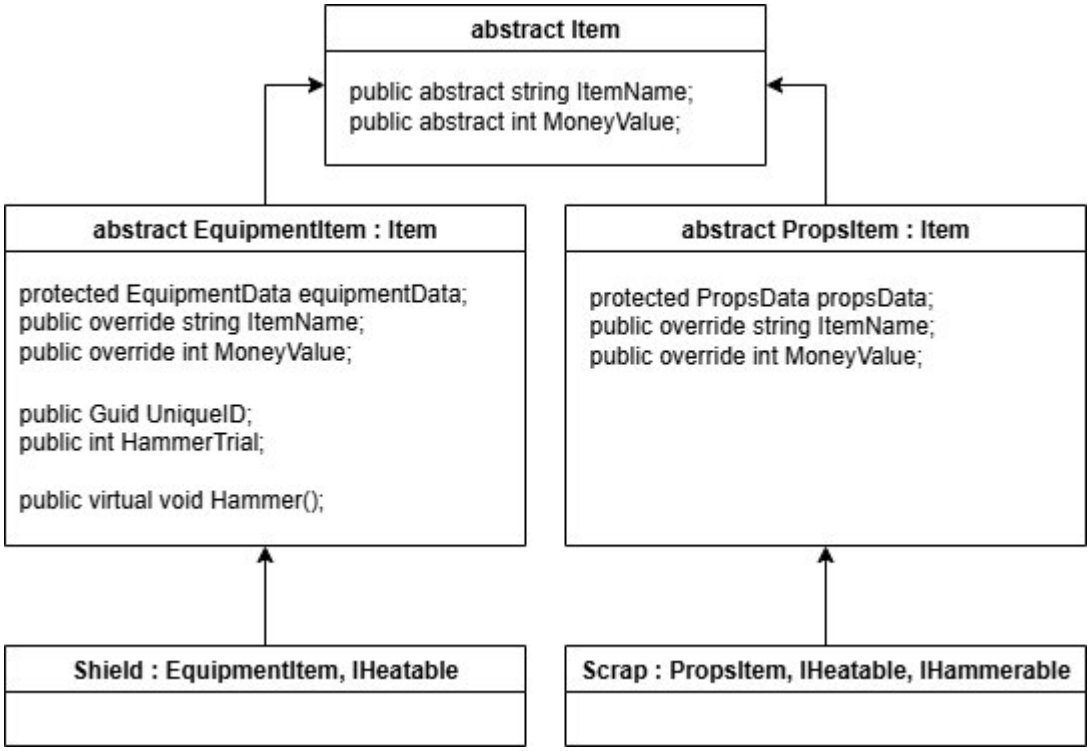
실내

시작화면에서 뒤로 돌면 보이는 대장간 내부.

해당 게임의 거의 모든 것은 게임이 시작하면 위치하는 실내에 모여있다. 코드 구현부 자체는 복잡한 편은 아니었고, 아이템이 들고 있는 **데이터를 설계하는 단계에서 시간을 조금 썼다**. 근본이 VR 게임이다보니 *VR 컨트롤러를 통한 상호작용이 주는 만족감에* 어느정도 기대게 되는 감이 있는데, Unity가 미리 마련한 **XR 프레임워크**는 습작에 **바로 사용하기에 충분**한 정도여서 설계에 시간을 오롯이 쓸 수 있었다.

장비 공통으로 뺄 지 고민했는데, 작업을 실내 씬에서 하니 여기에 놨다

실내 로직의 과반을 차지하는 파트이다. 게임에 **장비와 소모품 두 종류**의 아이템이 존재하고, 둘이 가지는 데이터 중 일부는 공유하는 바가 있어서 **상속 구조**를 통해 동일 기능을 제공하면서 코드를 간소화하기로 결정했다.



아이템과 그 하위 클래스들의 상속 관계

각 아이템들마다 **취할 수 있는 행동**은 **인터페이스**Interface 를 주고 마지막에 구현했다. 예외적으로, 망치질할 수 있는IHammerable 성격도 인터페이스로 나눌까 고민했었는데, 게임 자체가 **'강화' 를 표방**하고 있는 만큼, 강화는 **망치로 두들기는 게 제 맛**일거라 생각, 모든 장비는 강화를 할 것이므로 **추상 클래스 단위에 정의**하기로 결정했다.

장비의 단계별 스텟은 **Unity에서 제공하는 ScriptableObject**로 추가해주었다. 바뀔 일이 딱히 없는 정적 데이터다보니 빠르게 작업하고자 사용했는데, 근본은 **C#의 클래스와 다르지 않아**서 참조해 사용할 수도 있고, Inspector 상에서 빠르게 값을 입력해 데이터 객체를 만들기도 편리했다.

ScriptableObject가 원본 클래스, 그리고 만들어 둔 데이터 객체들은 해당 클래스의 복제품을 만들고 작성한 값을 대입하는 식으로 사실상 **메모리에 하나의 클래스만 올라가게** 구성한다고 한다.

화로

화로에 올려놓아야 달궈지고 강화를 시도할 수 있다.

화로Coals의 로직은 공통에서 언급한 **온도관리자의 것을 가져와 역행동**을 하게 뒤집었다. 풀어서 말하자면, 데워질 수 있는 오브젝트들이 화로에 닿으면 화로의 **컴포넌트를 구독**하게 만들고, 화로는 지속적으로 구독한 대상들의 **온도를 상승시키기만** 하는 식이다.

프로젝트 자체가 습작이라 가벼워 자칫 지나칠 수도 있지만, Unity의 Update() 생명주기Lifecycle 에 너무 많은 부담을 지우면 자원 소모를 감당하기 힘들어 렉lag 이 생기지 않겠냐는 판단이었다. 더욱이 일반적인 **게임 PC보다 사양이 낮은 편인 VR 기기**다보니 선수에 방지하자는 의도도 있었다.

```
// Register
void OnTriggerEnter(Collider other)
{
    other.gameObject.TryGetComponent<IHeatable>(out var heatable);
    if (heatable != null && !_heatables.Contains(heatable))
    {
        _heatables.Add(heatable);
    }
}

// Unregister
void OnTriggerExit(Collider other)
{
    other.gameObject.TryGetComponent<IHeatable>(out var heatable);
    _heatables.Remove(heatable);
}
```

온도관리자 컴포넌트에도 같은 내용이 있다

망치

망치를 두들긴 순간의 화면

망치Hammer는 이 게임에서 손에 들고 휘두를 만한 오브젝트 중 **아이템이 아닌**, 강화를 위한 도구이다. 그러다보니 Item 클래스를 따로 **상속받을 필요가 없어** 별도의 모노비헤이비어MonoBehaviour 스크립트를 짜게 됐다. 구현한 코드의 대부분은 망치가 망치질 할 수 있는 오브젝트에 닿았을 때 **화면과 소리 효과를 표현**하는 기능에 쓰였다.

고철

```
public class Scraps : PropsItem, IHeatable, IHammerable
```

객체-지향 프로그래밍을 접하고서 개인적으로 마음에 들었던 부분인 상속

고철Scrap은 소모품 아이템으로 분류했다. 이 게임에서 고철의 용도는 강화할 장비가 하나도 남지 않았을 때 첫 단계 장비를 얻을 수 있는 일종의 **페일세이프**failsafe 역할이다. 1단계 장비로의 강화 성공 확률이 100%라는 점과 소모품이므로 최대 보유 수량이 정해져 있단 것을 빼면 **장비처럼 강화가 진행**되어야 하기 때문에 예외적으로 고철 클래스에 망치질 할 수 있는 인터페이스를 붙여 강화 기능을 따로 구현해주었다.

코멘트

1. 망치가 망치질 할 수 있는 오브젝트와 충돌할 때- 강체Rigidbody 컴포넌트를 이용해 -가속력을 계산해 효과가 발동하게 했다면 더 자연스럽지 않았을까 생각한다. 현재는 단순히 충돌체 만을 고려하기 때문에 소위 옷깃에 바람이 스치듯 닿기만 해도 효과가 발생한다.

2. ScriptableObject를 사용하면서 불편했던 점은 내부의 데이터에 변동이 있을 때- 더 정확히는 기존에 있던 필드가 사라지거나, 입력한 값이 바뀔 때 -이미 작성해 둔 데이터 객체 파일들의 값이 초기화된다는 점이다. 없는 필드를 찾아서 붙여넣으라고 할 수는 없는 노릇이니 어쩔 수 없다지만 마음에 안드는 부분이라는 점은 바뀌지 않는다.

실외

XR 프레임워크가 제공하는 Climb이 어느 정도로 동작할 수 있는지 궁금해 동선을 복잡하게 짰다..

실외는 앞서 언급하였듯이 기능이랄 게 딱히 없다. 실외의 목적이 **Unity**의 **Terrain Tool**을 **구경**해보고 싶다는 것이었고, 그 부분 만큼의 목적은 달성했다. 다만, 머리를 비우고 지형을 깎다 보니 VR의 벽타기Climb 상호작용으로 장난치기 좋을 것 같은 구간이 보였고, 그걸 활용한 정도이다.

버섯

이 버섯을 대장간으로 가져가 쓰면 된다

버섯Mushroom 은 그 벽타기 장난을 끝까지 응해주었을 때 얻을 수 있는 보상으로 준비하였다. 소모품으로 분류되는 버섯은 그 단독으로썬 아무 역할도 하지 못하지만, **최대 온도까지 데워졌을 때** 주변의 **충돌을 감지**해 연관되는 오브젝트와 상호작용을 하고 난 뒤, (고온에서 버티는 버섯이 없을 것이므로) 파괴된다.

현재 화로와 상호작용이 가능하며, 최대 온도에 도달했을 때 주변에 화로가 충돌했다면, 화로의 **색을 바꾸고** 강화 **확률에 보정치**를 넣어주는 기능을 한다.

코멘트

1. Terrain Tool 을 사용하면서 나무와 잔디를 심었는데, 이런 요소들이 누적되면 기기에 부하를 줄 것이라 판단하여 의도적으로 카메라 거리를 줄이고, Unity 내장 안개 효과를 주었다. 문제라면, Unity 내장 안개 효과가 VR 기기에서 적용되지 않는다는 점인데, 이 부분은 외부 에셋을 쓸 게 아니라면 셰이더를 손보거나 내가 필요한 안개 셰이더를 직접 구현해야 할 것 같다.

남기는 말

인벤토리 로직에 대한 검수 및 **정비**와 **인벤토리 UI 구현**을 끝마치고 나면 이 습작을 놓아주려고 (정확히는 유지보수만 하려고) 한다. 새로운 것을 해보고 싶고, 그래야 경험치가 조금이나마 더 쌓이지 않겠냐는 생각이다.

보통의 포트폴리오라면 꼼꼼한 이미지와 상세한 설명이 동반되어야 할텐데, **본인이 그런 재주가 없다** 보니 **부득이하게 글로** 남기게 되었다. 그럼에도 불구하고, 긴 글을 끝까지 읽어준 독자들에게 감사의 인사를 표한다.