3D 포트폴리오

Shooting Game

RPG Game

김 도 영

Shooting Game

목 차

제작 정보

미사일 예상 경로 그리기

최적화를 위한 Object Pool

여러 상태를 제어하기 위한 State Pattern

순차적인 실행을 위한 Coroutine 활용

점수를 저장하기 위한 PlayerPrefs

제작 정보

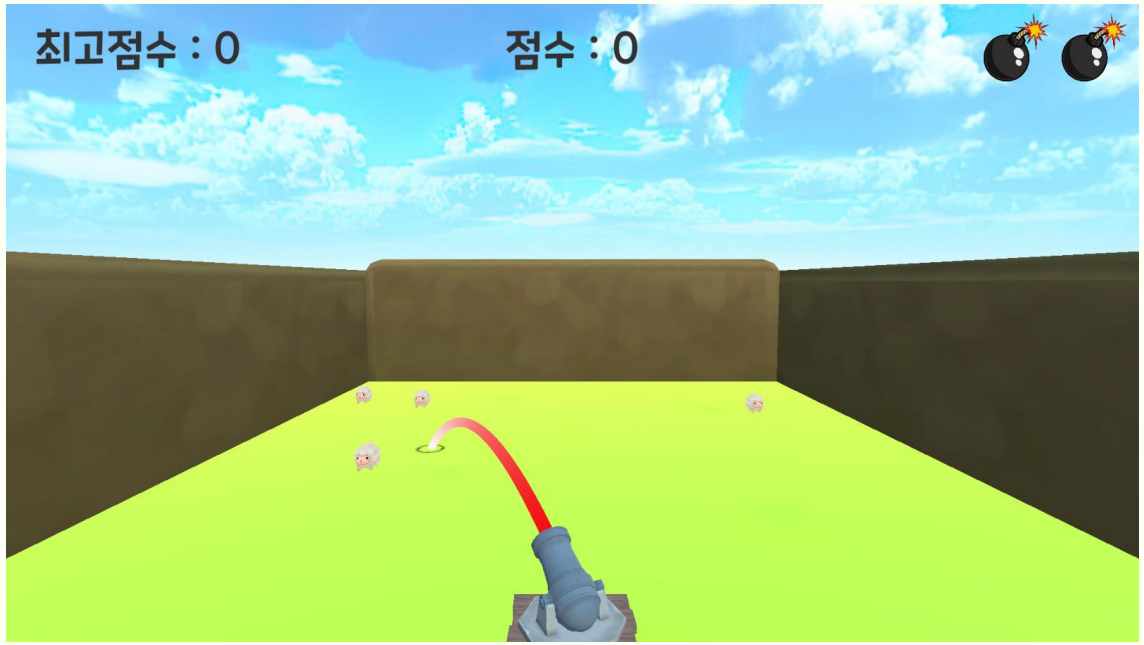
슈팅 게임

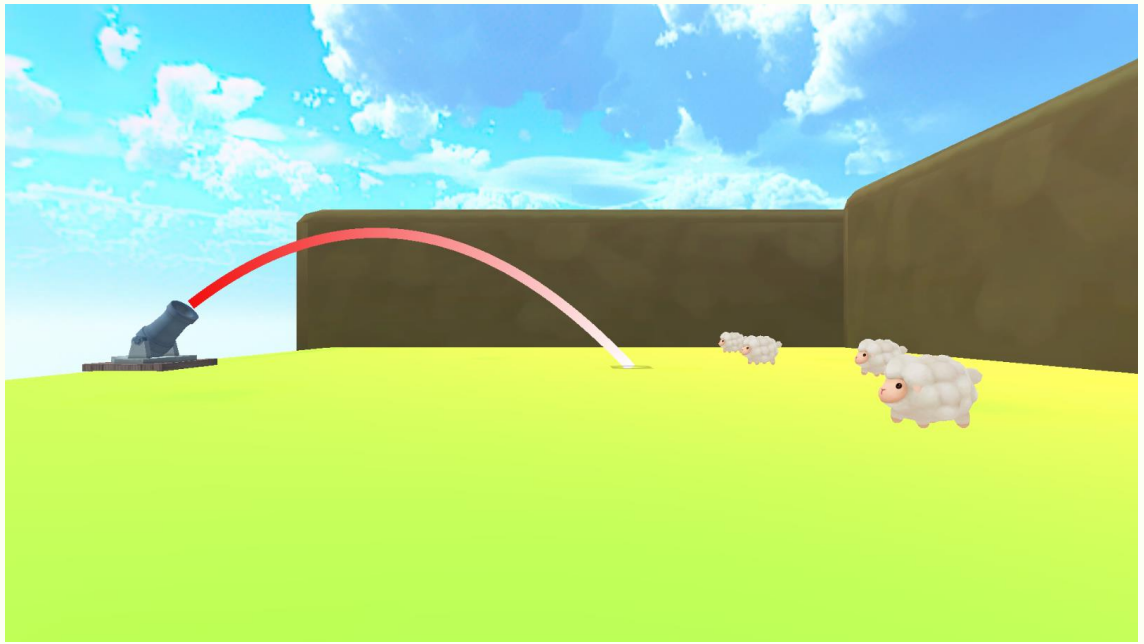
장르 : 슈팅

게임 방식 : 적의 숫자를 15마리 이하로 유지

개발 툴 : C#, Unity, Github

미사일 예상 경로 그리기





Line Renderer를 사용하여 각 포인트들로 선을 그렸습니다.

미사일의 운동방식이 등가속운동이므로 각 포인트마다

그 시간때의 속도를 더하여 각 포인트를 도출 후 연결하였습니다.

미사일 예상 경로 그리기

```
for (int i = 1; i <= resolution; i++)
{
    float simulationTime = i / (float)resolution * 10f;

    Vector3 displacement = velocity * simulationTime + Vector3.up * Physics.gravity.y * simulationTime * simulationTime / 2f;

    Vector3 drawPoint = barrel.transform.GetChild(0).gameObject.transform.position + displacement;
    index.Add(drawPoint);
    previousDrawPoint = drawPoint;

    if (previousDrawPoint.y <= 0.0f)
    {
        line.positionCount = index.Count;
        line.SetPositions(index.ToArray());
        // 목표지점 그리기
        mainPoint.transform.position = new Vector3(previousDrawPoint.x, 0.1f, previousDrawPoint.z);

        return;
    }
}
```

$$s = vt + \frac{1}{2}at^2$$

s = displacement

v = velocity

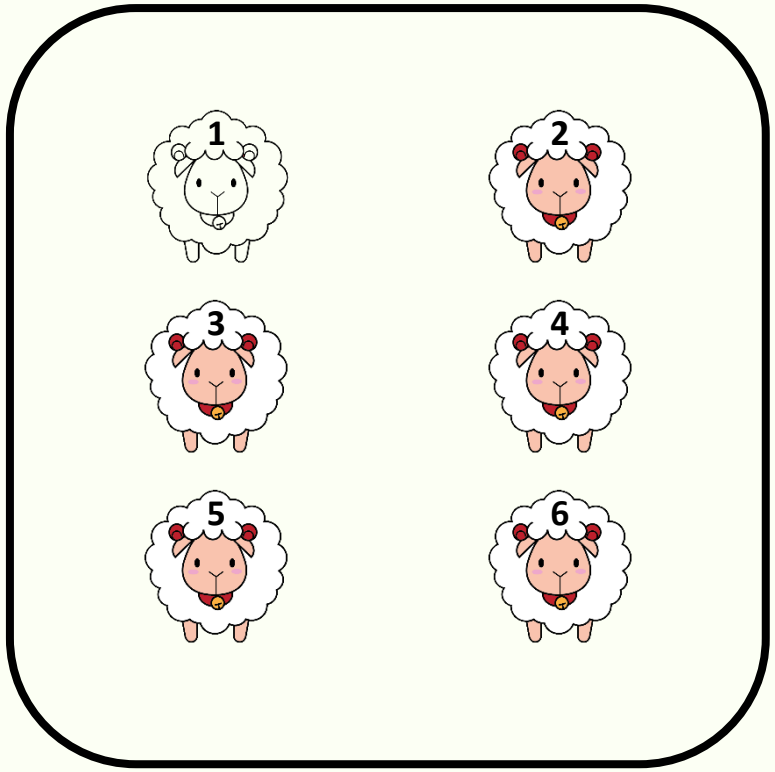
t = simulationTime

a = Vector3.up * Physics.gravity.y

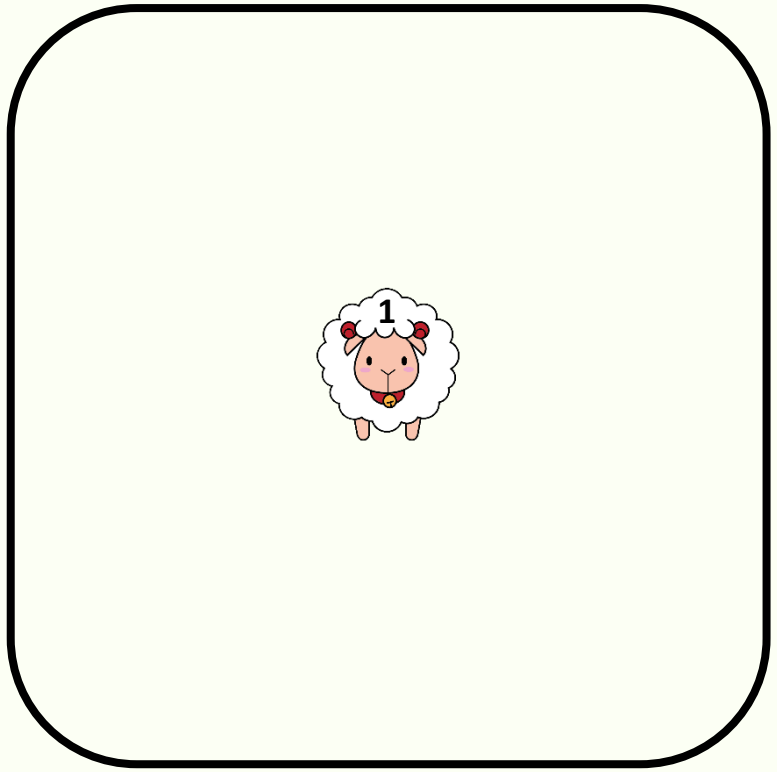
- 등가속 운동의 속도 구하는 방정식을 코드화
- 포신의 방향/파워를 활용하여 미사일 초기속도 도출
- 도출된 초기속도에 등가속 운동을 적용, 드로우포인트 도출

최적화를 위한 Object Pool

Object Pool



Play Scene

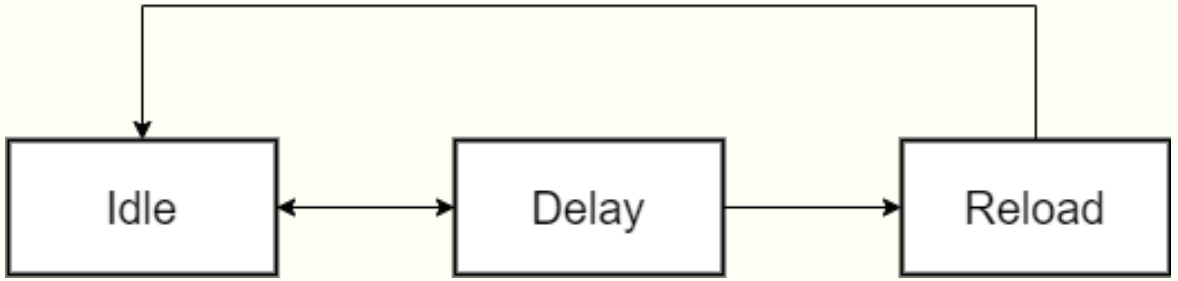


싱글턴으로 오브젝트풀을 만들어서 큐에 오브젝트를 미리 생성했습니다.

오브젝트가 필요할 때 풀에서 가져오는 방식으로 하여 객체의 생성으로 인한 성능저하를 개선했습니다.

여러 상태를 제어하기 위한 State Pattern

```
public enum State
{
    Idle,
    Delay,
    Reload
}
```



Idle	발사가 가능한 상태입니다. 포탑을 상하좌우로 움직일 수 있습니다.
Delay	발사 후의 상태입니다. 발사는 불가능하지만 포탑을 움직일 수는 있습니다.
Reload	모든 미사일을 소진한 상태입니다. 모든 행동이 제한됩니다.

순차적인 실행을 위한 Coroutine 활용

호출

```
void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Missile")
    {
        gameObject.GetComponent<Rigidbody>().AddExplosionForce(5000.0f, other.gameObject.transform.position, 5.0f, 1000.0f);
        StartCoroutine("returnEnemy");
    }
}
```

폭발

실행

```
IEnumerator returnEnemy()
{
    GameObject.Find("GameManager").GetComponent<GameManager>().score += 10;
    yield return new WaitForSeconds(2.5f);
    ObjectPool.Instance.ReturnObject(gameObject);
    Rigidbody rigid = gameObject.GetComponent<Rigidbody>();
    rigid.velocity = new Vector3(0f, 0f, 0f);
    rigid.angularVelocity = new Vector3(0f, 0f, 0f);
    transform.rotation = Quaternion.Euler(new Vector3(0f, 0f, 0f));
}
```

점수

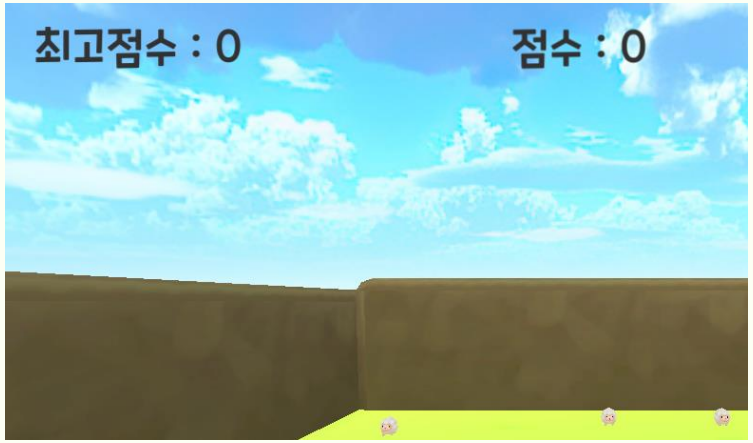
오브젝트 풀 반환

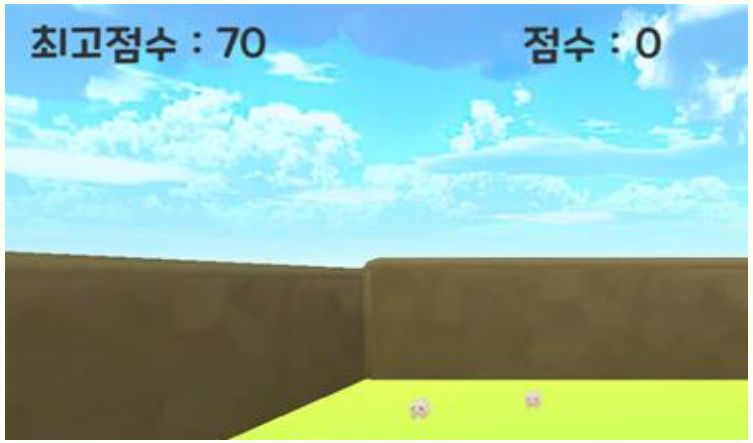
적이 죽었을때 Coroutine을 활용하여 순차적으로 '폭발 -> 점수 -> 오브젝트 풀 반환' 실행하도록 구현하였습니다.

점수를 저장하기 위한 PlayerPrefs

```
Public void SetData(int bestScore)
{
  If (최고점수가 갱신되었는가?)
  {
    최고점수를 저장
  }
}

Public int GetData()
{
  If (최고점수가 존재하는가?)
    최고점수를 반환
  else
    0을 반환
}
```





최고점수는 PlayerPrefs를 이용해 컴퓨터에 저장하여

게임시작시 데이터를 불러오도록 하였고,

게임이 끝났을 때 최고점수에 변동이 있을 때만 데이터를 저장하도록 하였습니다.

RPG Game

목 차

INVENTORY

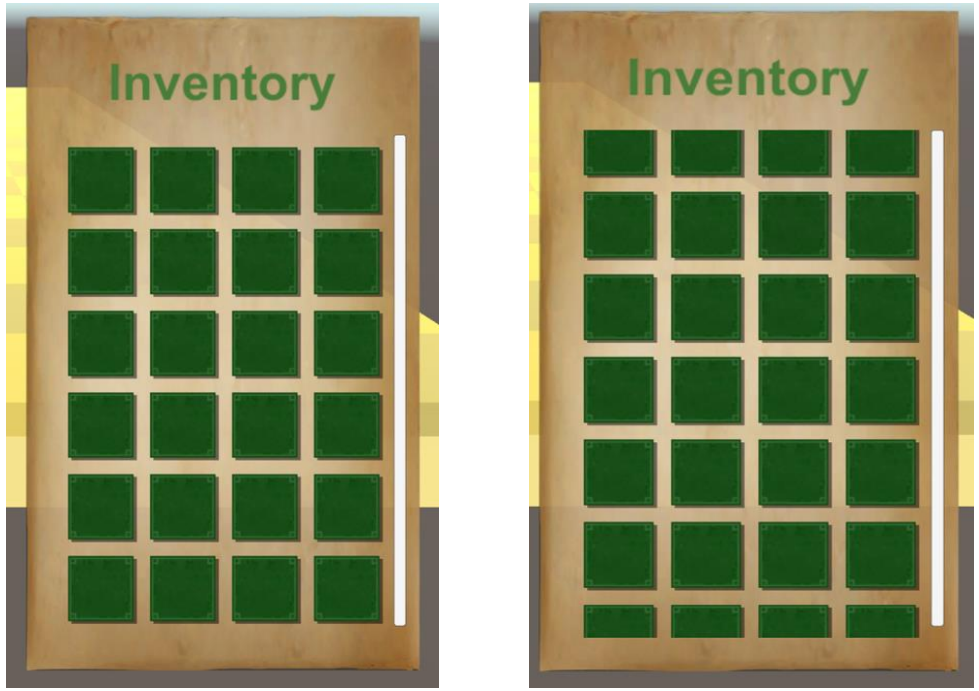
※ 기본 상태

※ 아이템을 먹은 후

- 모든 아이템마다 각 각의 아이템의 정보를 넣어주었습니다.
- 충돌된 아이템의 정보를 Inventory의 List에 넣어주었습니다.
- 충돌될 때마다 Inventory의 정보와 이미지를 갱신하도록 하였습니다.

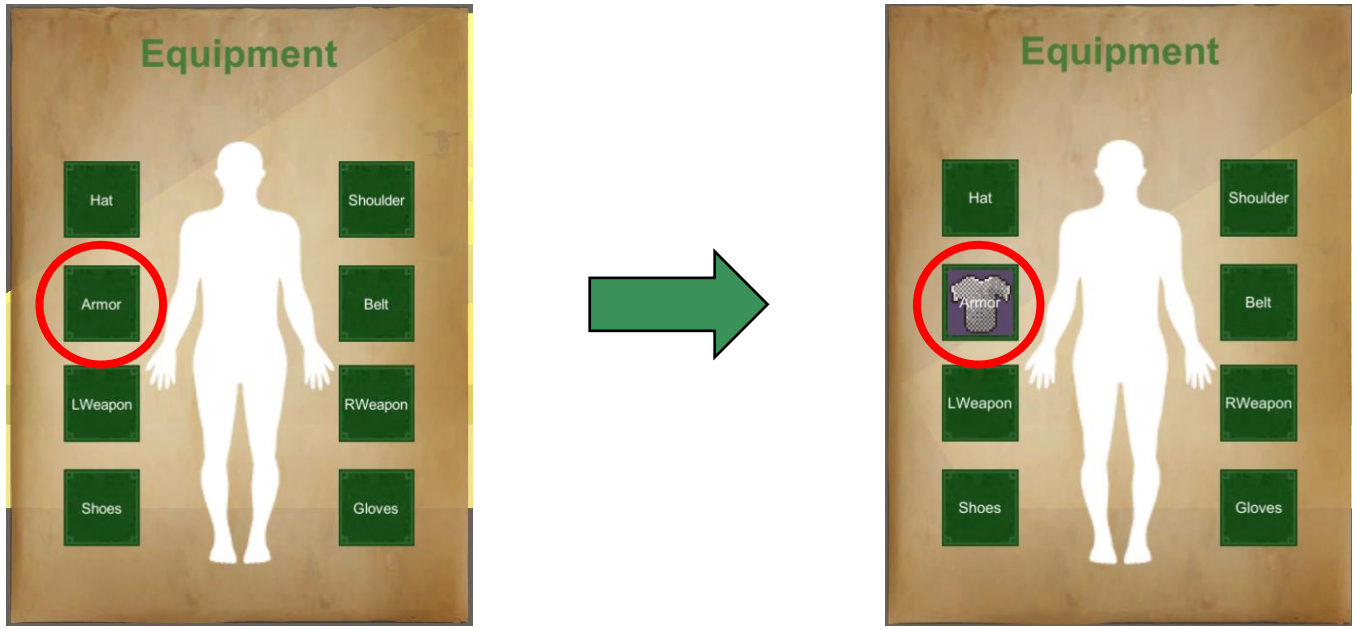
INVENTORY

- Grid Layout Group 를 사용하였습니다.
- 인벤토리 1칸을 Prefabs화 하여 Grid Layout Group 적용을 쉽게 하였습니다.
- Grid Layout Group의 기능 사용하여 Inventory의 Scroll을 구현하였습니다.
- INVENTORY는 정보를 쉽게 수정하기 위해 List를 사용하였습니다.



※ Scroll 구현

EQUIPMENT



- 현재 활성화된 장비를 확인 하기위해 “Tag”를 사용하여 원하는 오브젝트에 접근하는 것을 목표
- 자식 오브젝트에 중복된 “Tag”로 인해 접근이 어려운 것을 확인
- 접근을 위해 각각의 아이템과 장비칸에 ID와 번호를 저장한 후 INTERFACE를 활용하여 반환함
- ID를 사용하여 같은 부위의 오브젝트에 접근과 확인 후 번호를 이용하여 파츠 변경을 하였습니다.

EQUIPMENT

1. Tag 사용 실패

```
player = GameObject.FindWithTag("Player");
//for(GameObject.FindGameObjectsWithTag(this.gameObject.tag))
string tag = this.gameObject.tag;
equip = player.transform.Find(tag);
```

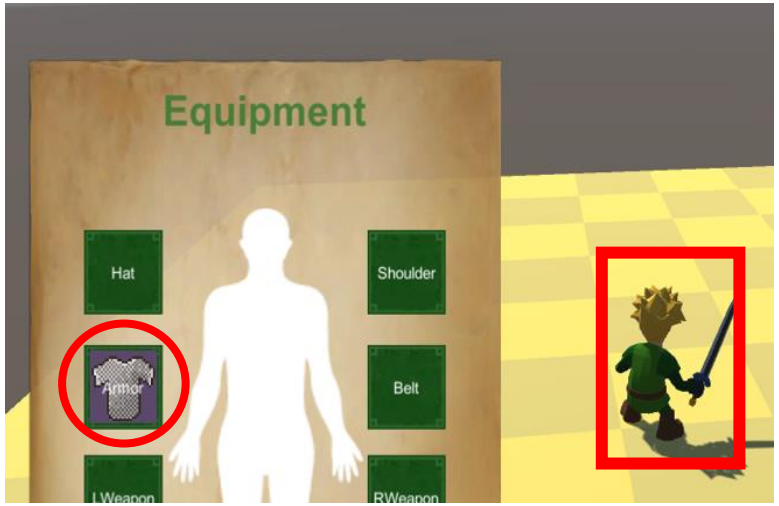
2. Interface 사용

```
public interface IEquipment
{
    참조 1개
    int IID { get; }
    참조 1개
    int INumber { get; }
    참조 3개
    bool IActivate { get; }
}
```

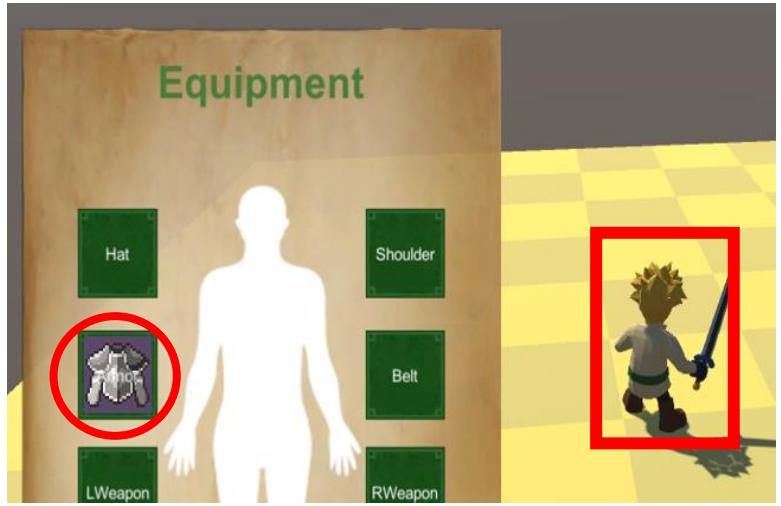
3. Interface로 만든 IActiveate를 이용하여 접근

```
GameObject equip = player.transform.GetChild(1).gameObject;
for (int i = 0; i < equip.transform.childCount; i++)
{
    if (equip.transform.GetChild(i).gameObject.GetComponent<Cloth>().IActivate == true)
    {
        item.ID = equip.transform.GetChild(i).gameObject.GetComponent<Cloth>().ID;
        item.Number = equip.transform.GetChild(i).gameObject.GetComponent<Cloth>().Number;
        item.itemName = equip.transform.GetChild(i).gameObject.GetComponent<Cloth>().item.itemName;
        item.itemImage = equip.transform.GetChild(i).gameObject.GetComponent<Cloth>().item.itemImage;
        item.itemType = equip.transform.GetChild(i).gameObject.GetComponent<Cloth>().item.itemType;
        break;
    }
}
```

장비 교체



※ 현재 착용한 장비



※ 착용 장비를 교체

- 현재 활성화된 오브젝트를 Interface를 통해서 확인할 수 있게 하였습니다.
- 확인된 오브젝트의 이미지를 장비창에 나타냈습니다.
- 선택된 아이템의 ID와 번호를 통해서 같은 파츠를 찾고 교체를 하도록 하였습니다.

장비 교체

※ 장비 교체 전

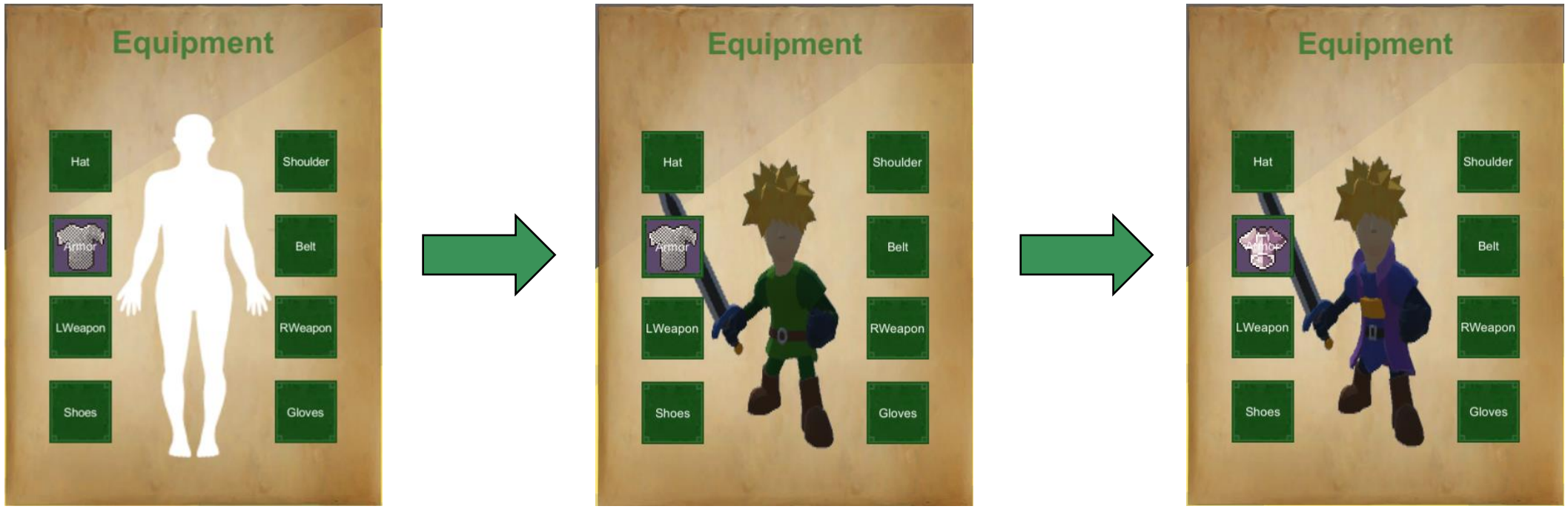
※ 장비 교체 후

- 장비 교체 시 ID를 통해 Player의 자식 오브젝트에 접근 하였습니다.
- 아이템의 번호로 현재 활성화된 오브젝트를 꺼주고 바뀐 오브젝트를 활성화했습니다.

장비 교체

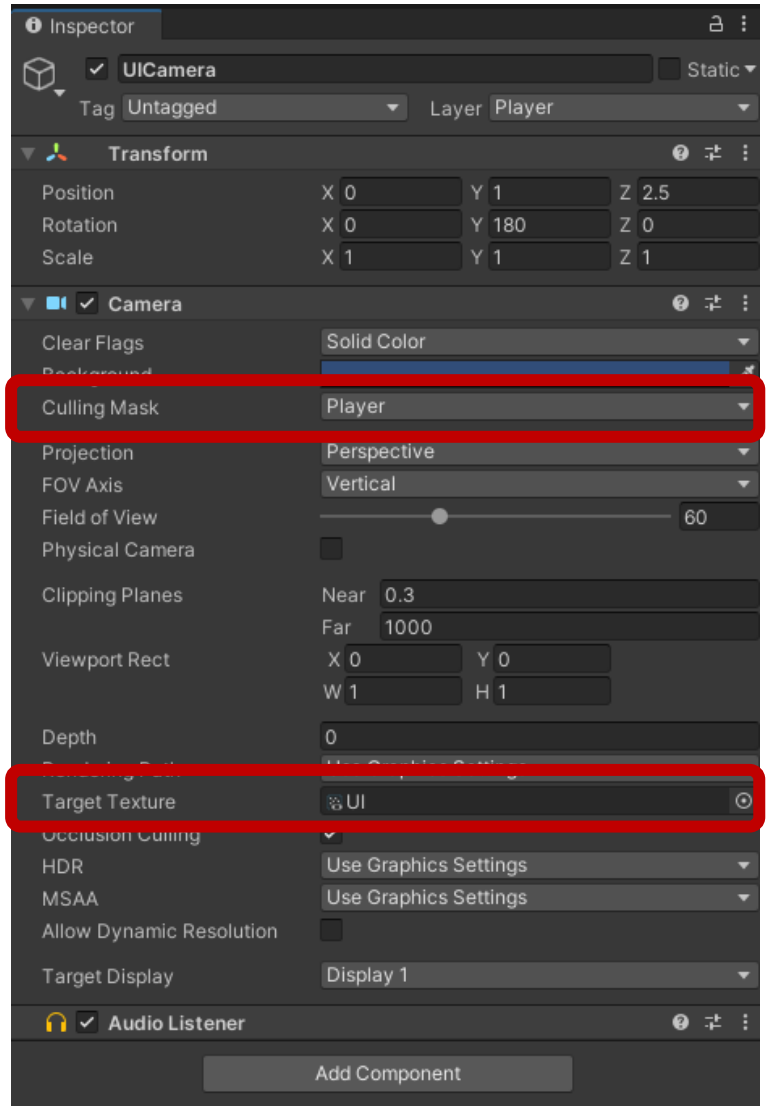
- 아이템을 사용했을 때 사용된 아이템이 장착되고 기존 장착중인 아이템은 새로 장착된 아이템이 있던 슬롯으로 들어갑니다.

Render Texture

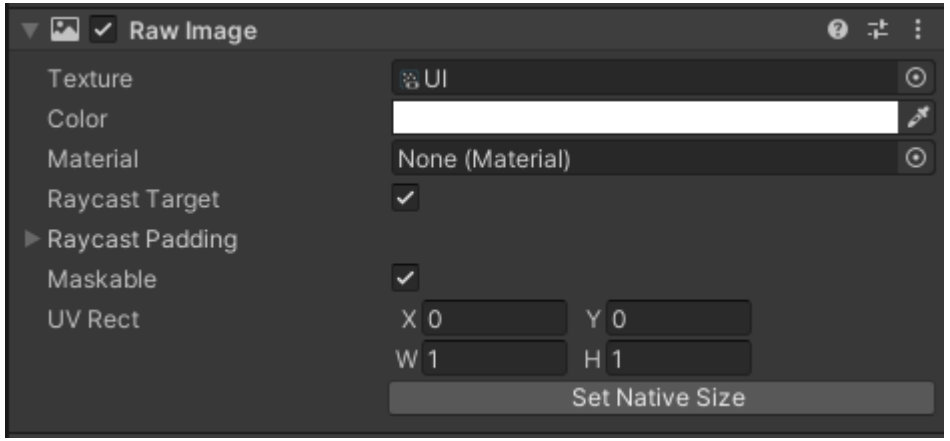


- Render Texture를 만들고 UI카메라를 만들어서 목표하는 오브젝트를 바라보게 했습니다.
- Raw Image를 사용하여 Render Texture를 장비 UI에 띄웠습니다.

Render Texture



1. 선택한 Layer만 Render합니다.



※ UI용 Raw Image

2. Target Texture에 Render Texture 넣으면 선택한 카메라가 선택된 Layer만 Render Texture에 저장합니다.

상 점

Collider 범위

상점 TextUI

Player와 NPC 서로 통과하지않고
Player가 범위내에 들어왔을 때 상호작용을 위해서 2개의 Collider를 사용하였습니다.

2번째 Collider에 isTrigger를 사용하여 Player의 접근을 확인하였습니다.

NPC에 WorldSpace로 된 Canvas를 넣어
카메라를 향해 바라보도록 하였습니다.

상 점

상점 UI

Prefabs 구성

Grid Layout Group 를 사용하여
추후 아이템 목록이 추가 / 삭제
될 때 용이하도록 하였습니다.

```
Text textObj = gameObject.transform.parent.GetChild(1).GetComponent<Text>();
textObj.text = item.itemName;
Image imageObj = gameObject.transform.parent.GetChild(2).GetComponent<Image>();
imageObj.sprite = item.itemImage;
```

각 Prefabs에 아이템의 정보를 넣었습니다.

아이템의 이름과 이미지를 UI에 반영하였습니다.

상 점

상점을 구현하면서 Inventory 접근을 위해 Singleton을 Inventory에 적용하였습니다.

상점의 아이템을 Inventory List에 추가할 수 있도록 하였습니다.

아이템 드랍

```
Unity 메시지 | 참조 0개
private void Start()
{
    tempPosition = transform.position;
    rigidbody.MovePosition(transform.position + Vector3.up);
    StartCoroutine(Position());
}

참조 1개
IEnumerator Position()
{
    yield return new WaitForSeconds(0.5f);
    this.GetComponent<Rigidbody>().isKinematic = true;
    this.transform.position = tempPosition;
}
```

- 아이템이 상자에서 튀어나오는 연출
- 아이템이 바닥을 통과하는 문제를 막기 위해 Coroutine을 활용하여 일정시간 후 isKinematic를 활성화되도록 하였습니다.

아이템 획득

```
private void OnTrigger(Collider other)
{
    if (other.CompareTag("Player"))
    {
        if (Input.GetKeyDown(KeyCode.G))
        {
            Inventory.instance.InItem(this);
        }
    }
}
```

- 아이템 범위 내에 플레이어가 접근 시, 상호작용 키가 활성화 되도록 하였습니다.
- 플레이어가 상호작용 키를 누를 시 아이템이 인벤토리에 들어가고 필드에서는 삭제되도록 구현하였습니다.