프로젝트 포트폴리오

작성자 : 조건상

목차

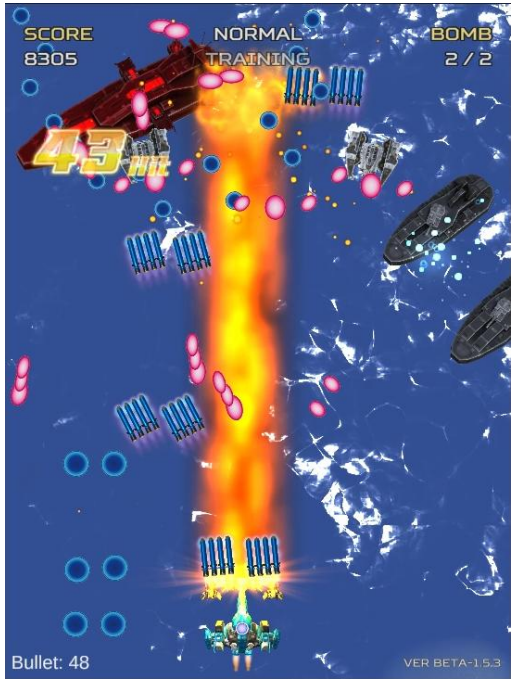
- 프로젝트 개요
- 리플레이 시스템
- 충돌 처리
- 구현 결과

프로젝트 개요

- **프로젝트** : Shooting Game
- **Github 주소** : https://github.com/jeffjks/UnityShootingGame
- **개발 인원** : 1인 개발
- **프로젝트 개요** : 유니티를 사용하여 구현한 탄막 슈팅 게임입니다.
  Deterministic한 리플레이 시스템과 이를 위한 자체적인 충돌 처리 시스템을 구현한 것이 특징입니다.

프로젝트 개요

- 3D 그래픽을 사용한 탄막 슈팅 게임

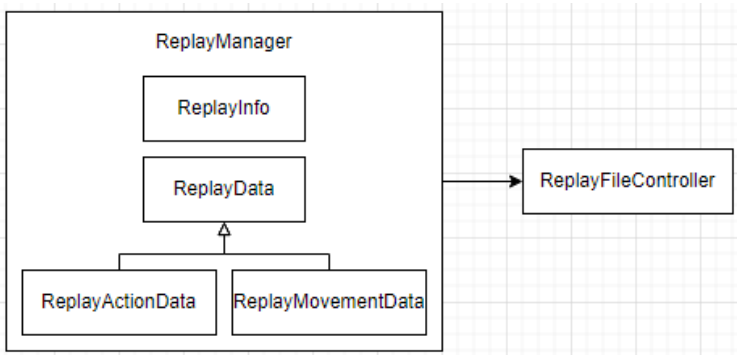


리플레이 시스템

Input 기반 방식	상태 저장 방식
• 사용자의 Input 데이터만을 받아 이를 사용하여 리플레이 실행 • 작은 리플레이 파일 크기 • 정교한 Determinism한 시스템 필요	• 매번 모든 오브젝트들의 상태 저장 • 큰 리플레이 파일 크기 • 원하는 리플레이 시점으로 이동하기 쉬움

- 매번 오브젝트들의 상태를 저장하기에는 오브젝트 수가 너무 많고 리플레이 파일 길이가 김 (리플레이 파일 길이는 최대 30분 정도)
- 원하는 리플레이 시점으로 이동하는 기능은 크게 필요하지 않음
  -> Input 기반 리플레이 시스템 채택

리플레이 시스템 - 주요 클래스



- **ReplayManager**: 유저로부터 입력받은 데이터를 리플레이 파일에 쓸 데이터로 변환하거나, 반대로 리플레이 파일로부터 읽은 데이터를 리플레이 재생에 필요한 형태로 변환하는 클래스
- **ReplayInfo**: 리플레이 파일의 버전, 시드 등의 기본 정보를 나타내는 클래스
- **ReplayData**: 리플레이에 기록될 키 입력(Input) 데이터
  ReplayActionData: 움직임 이외의 행동에 대한 입력 데이터
  ReplayMovementData: 움직임에 대한 입력 데이터
- **ReplayFileController**: 리플레이 데이터를 직접 파일에 쓰거나 읽는 동작을 구현한 클래스

리플레이 시스템 - 파일 관리

- BinaryFormatter를 사용하여 리플레이 데이터를 Read/Write
- AES 알고리즘으로 암호화
- ReplayDataType 값을 Input 데이터 맨 앞에 붙이는 방식으로 Input 데이터의 종류 구분
- 리플레이 파일 맨 앞에는 Random Seed값, 버전 정보 등의 정보 저장

리플레이 시스템 - Command 패턴

```
public interface IReplayInput
{
    public bool IsActive { get; set; } // 동시에 여러 Input 방지
    public int Frame { get; set; }

    public void RunData();
}
```

```
public static void ReadUserInput()
{
    if (_replayDataBuffer.Count < QueueMinCount)
    {
        ReadReplayDataToBuffer();
    }

    while (_replayDataBuffer.Count > 0 && _replayDataBuffer.Peek().Frame == CurrentFrame)
    {
        var replayData = _replayDataBuffer.Dequeue();
        replayData.RunData();
    }
}
```

- Command 패턴을 사용하여 리플레이 시스템 구현
- 실행할 작업을 IReplayInput 인터페이스로 추상화

리플레이 시스템 - Command 패턴

```cs
[Serializable]
public class ReplayMovementData : IReplayInput
{
    public bool IsActive
    {
        get { return isActive; }
        set { isActive = value; }
    }

    public int Frame
    {
        get { return frame; }
        set { frame = value; }
    }

    [NonSerialized] private bool isActive;
    private int frame;

    private byte inputMovement;

    public void RunData()
    {
        var moveVectorInt = GetMoveVectorData();
        _playerController.OnMoveInvoked(moveVectorInt);
    }
```

```cs
[Serializable]
public class ReplayActionData : IReplayInput
{
    public bool IsActive { get; set; }
    public int Frame { get; set; }

    private readonly KeyType keyType;
    private readonly bool isPressed;

    public ReplayActionData(int frame, KeyType keyType, bool isPressed)
    {
        Frame = frame;
        this.keyType = keyType;
        this.isPressed = isPressed;
    }

    public void RunData()
    {
        switch (keyType)
        {
            case KeyType.Fire:
                _playerController.OnFireInvoked(isPressed);
                break;
            case KeyType.Bomb:
                _playerController.OnBombInvoked(isPressed);
                break;
            default:
                Debug.LogError($"Unknown key type: {keyType}");
                break;
        }
    }
}
```

코드 위치: https://github.com/jeffjks/UnityShootingGame/blob/master/Assets/Scripts/Managers/ReplayManager.cs

리플레이 시스템 - Command 패턴

```
public static void ReadUserInput()
{
    if (_replayDataBuffer.Count < QueueMinCount)
    {
        ReadReplayDataToBuffer();
    }

    while (_replayDataBuffer.Count > 0 && _replayDataBuffer.Peek().Frame == CurrentFrame)
    {
        var replayData = _replayDataBuffer.Dequeue();
        replayData.RunData();
    }
}
```

리플레이 파일 읽기 관련 메소드

리플레이 파일의 input 데이터를
Queue<IReplayInput> _replayDataBuffer 에 저장하여
순차적으로 RunData() 함수로 작업 수행

```
public static void WriteUserMovementInput(Vector2Int inputVector)
{
    if (SystemManager.IsReplayMode)
        return;

    _movementContext.SetByteData(inputVector);
}

public static void WriteUserActionInput(KeyType keyType, bool isPressed)
{
    if (SystemManager.IsReplayMode)
        return;

    var actionContext = new ReplayActionData(CurrentFrame, keyType, isPressed);

    ReplayFileController.WriteBinaryReplayData(ReplayDataType.PlayerActionInput, actionContext);
}
```

리플레이 파일 쓰기 관련 메소드

WriteBinaryReplayData 함수를 호출하여
Input 데이터를 리플레이 파일에 저장
매개변수로 ReplayDataType 을 넘겨
Input 데이터 종류를 구분하여 저장

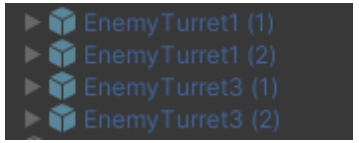
코드 위치: https://github.com/jeffjks/UnityShootingGame/blob/master/Assets/Scripts/Managers/ReplayManager.cs

리플레이 시스템 - Determinism 구현

- DefaultExecutionOrder를 사용하여 스크립트 실행 순서를 설정하여 일관된 결과가 나오도록 설정
  → 오브젝트의 이동과 충돌 체크의 순서에 따라 결과가 달라지는 문제 방지

```
[DefaultExecutionOrder(-101)]
public class PlayerController : MonoBehaviour
{
```

- Scene에 배치된 오브젝트의 로드 순서 설정
  → Random 함수의 실행 순서 보장



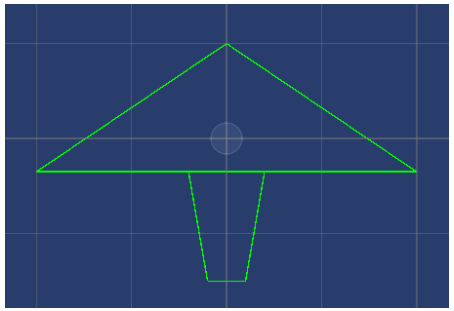
Scene에 배치된 순서대로 호출되지 않는다.

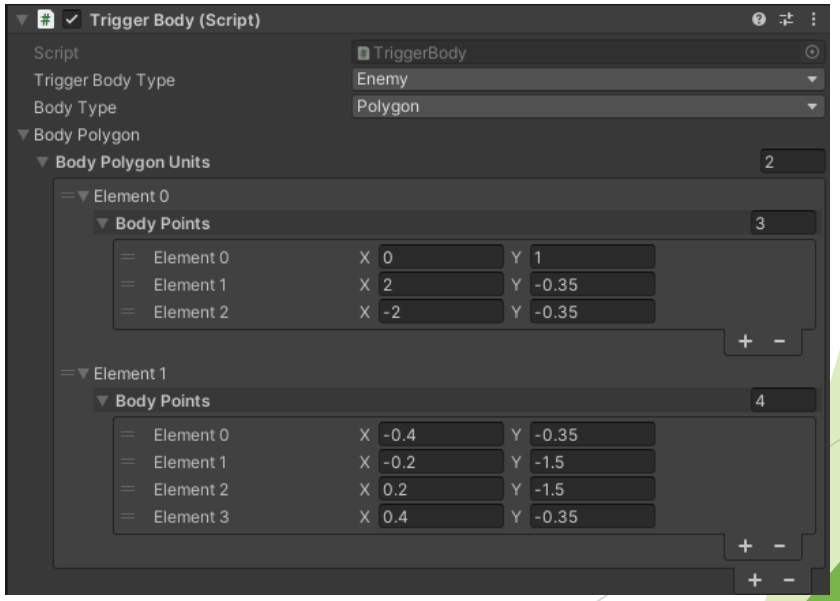
```
protected void InitEnemies()
{
    m_EnemySpawners.SetActive(true);

    for (int i = 0; i < (int) SystemManager.Difficulty + 1; i++)
    {
        LoadChildEnemyObject(m_EnemyPreloaded[i]);
    }
}
```

리플레이 시스템 - Determinism 구현

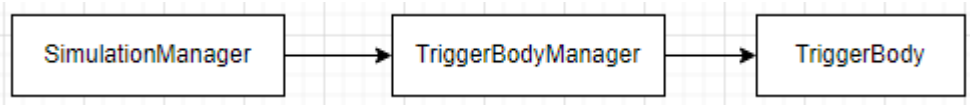
- 게임 특성상 시간 기반이 아닌 프레임 기반 시스템 채택
- 하지만 유니티의 Physics 시스템은 FixedUpdate 기반...
- Collider를 사용한 충돌 감지 부분에서 Determinism 하지 않는 문제 발생

=> TriggerBody라는 자체적인
충돌 처리 시스템을 구현하여 해결





충돌 처리 - 주요 클래스



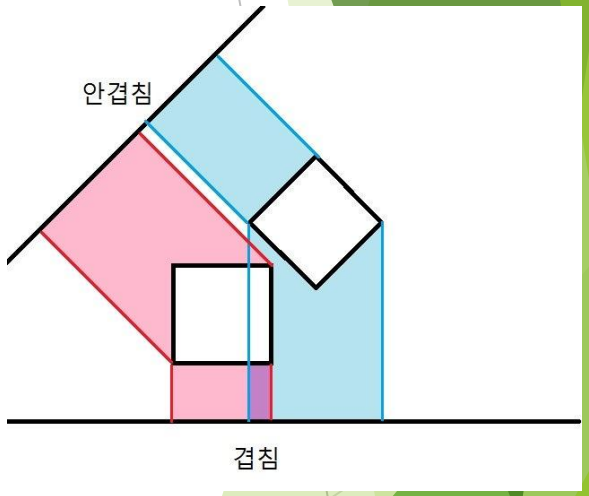
- **SimulationManager**: 충돌체(TriggerBody) 목록을 관리하고 충돌체간의 충돌 처리를 실행하는 클래스
- **TriggerBodyManager**: 충돌체의 타입에 따른 충돌 처리를 구현한 클래스
- **TriggerBody**: 충돌체. 원형 충돌체와 다각형 충돌체 타입을 구현

충돌 처리

- 다각형 모양의 TriggerBody와 원 모양의 TriggerBody 타입 구현
- 다각형 모양의 충돌 체크의 경우 **축 분리 이론(Separating Axis Theorem)**를 사용

```
private static bool CheckOverlapBodyUnit(BodyPolygonUnit bodyPolygonUnitA, BodyPolygonUnit bodyPolygonUnitB)
{
    var countA = bodyPolygonUnitA.m_BodyPoints.Count;
    var countB = bodyPolygonUnitB.m_BodyPoints.Count;

    for (var i = 0; i < countA + countB; ++i)
    {
        var currentEdge = i < countA
            ? bodyPolygonUnitA.m_BodyPoints[(i + 1) % countA] - bodyPolygonUnitA.m_BodyPoints[i]
            : bodyPolygonUnitB.m_BodyPoints[(i - countA + 1) % countB] - bodyPolygonUnitB.m_BodyPoints[i - countA];

        var axis = new Vector2(-currentEdge.y, currentEdge.x);
        axis.Normalize();

        // Axis에 대해 사영한 그림자의 범위 (min ~ max)
        ProjectBodyUnit(axis, bodyPolygonUnitA.m_BodyPoints, out var minA, out var maxA);
        ProjectBodyUnit(axis, bodyPolygonUnitB.m_BodyPoints, out var minB, out var maxB);

        if (IsIntersectDistance(minA, maxA, minB, maxB) == false)
            return false;
    }

    return true;
}
```

코드 위치:
https://github.com/jeffjks/UnityShootingGame/blob/master/Assets/Scripts/Managers/TriggerBodyManager.cs



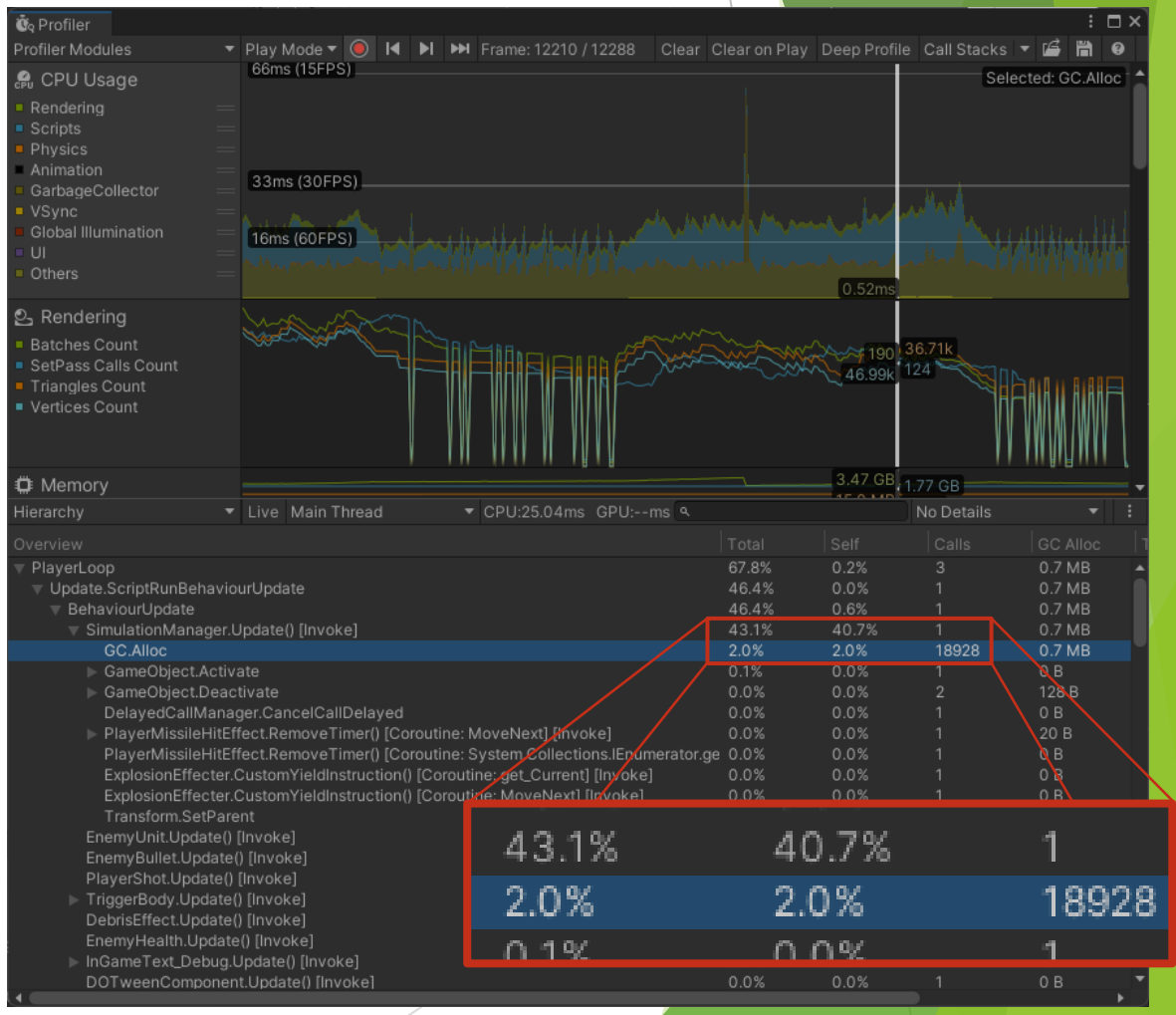
충돌 처리 - 최적화

- TriggerBody의 충돌 범위 좌표를 글로벌 좌표로 변환하는 과정에서 과도한 GC.Alloc Calls 발생

- 충돌관련 코드에서 과도한 CPU 점유율 발생

```
public BodyCircle TransformedBodyCircle => GetTransformedBody(m_BodyCircle);
public BodyPolygon TransformedBodyPolygon => GetTransformedBody(m_BodyPolygon);
```

가장 문제가 되었던 코드

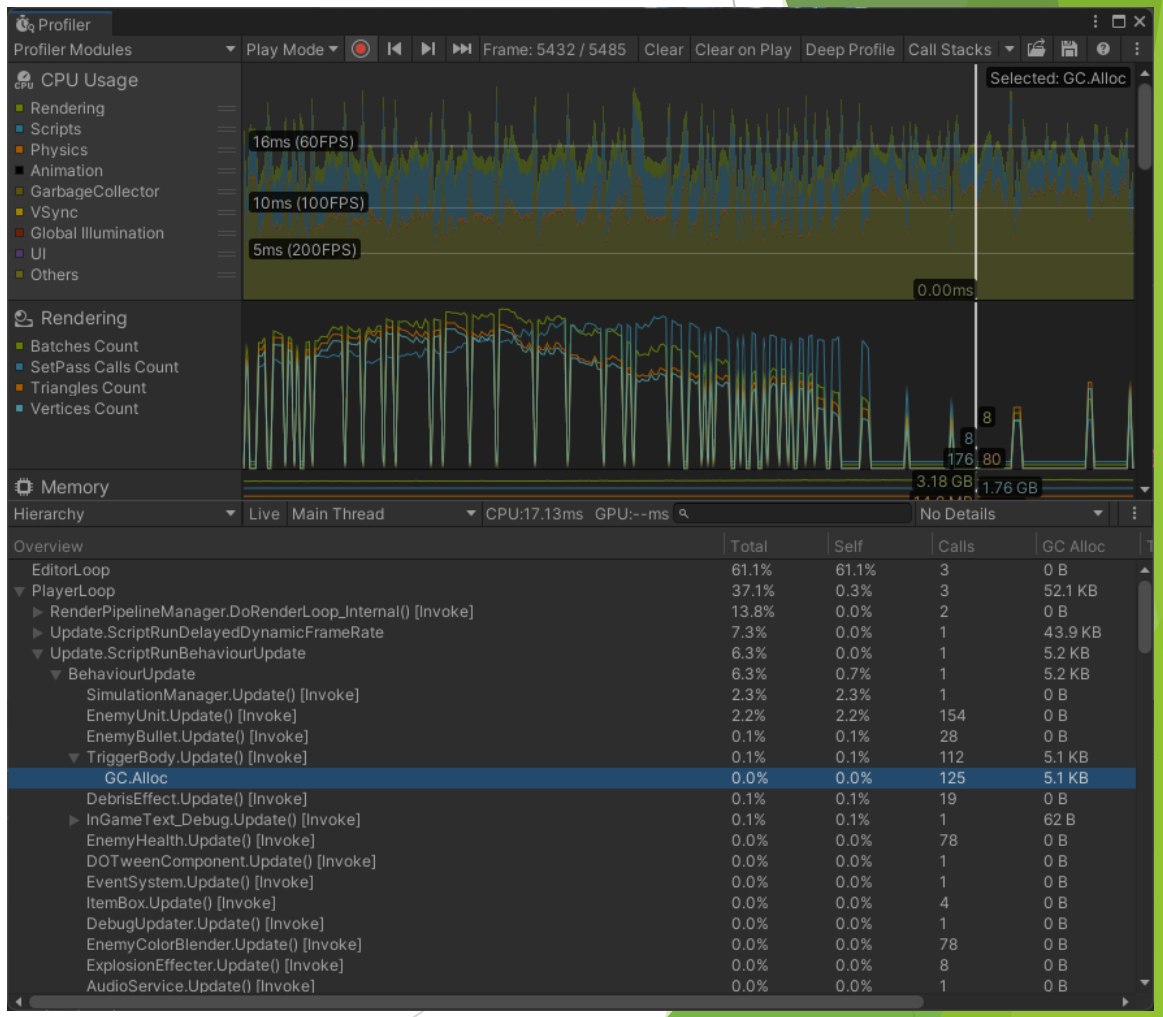


충돌 처리 - 최적화

- 해당 프로퍼티를 호출할 때마다 객체를 생성하지 않도록 수정 및 **캐싱** 사용
- Gc.Alloc Calls 횟수와 CPU 점유율이 획기적으로 감소

```
public BodyCircle TransformedBodyCircle
{
    get
    {
        if (_isTransformedBodyCircleDirty)
        {
            _cachedTransformedBodyCircle = GetTransformedBody(m_BodyCircle);
            _isTransformedBodyCircleDirty = false;
        }
        return _cachedTransformedBodyCircle;
    }
    private set
    {
        _cachedTransformedBodyCircle = value;
    }
}
```

코드 위치:
https://github.com/jeffjks/UnityShootingGame/blob/master/Assets/Scripts/TriggerBody.cs



충돌 처리 - 최적화

기타 최적화 이슈

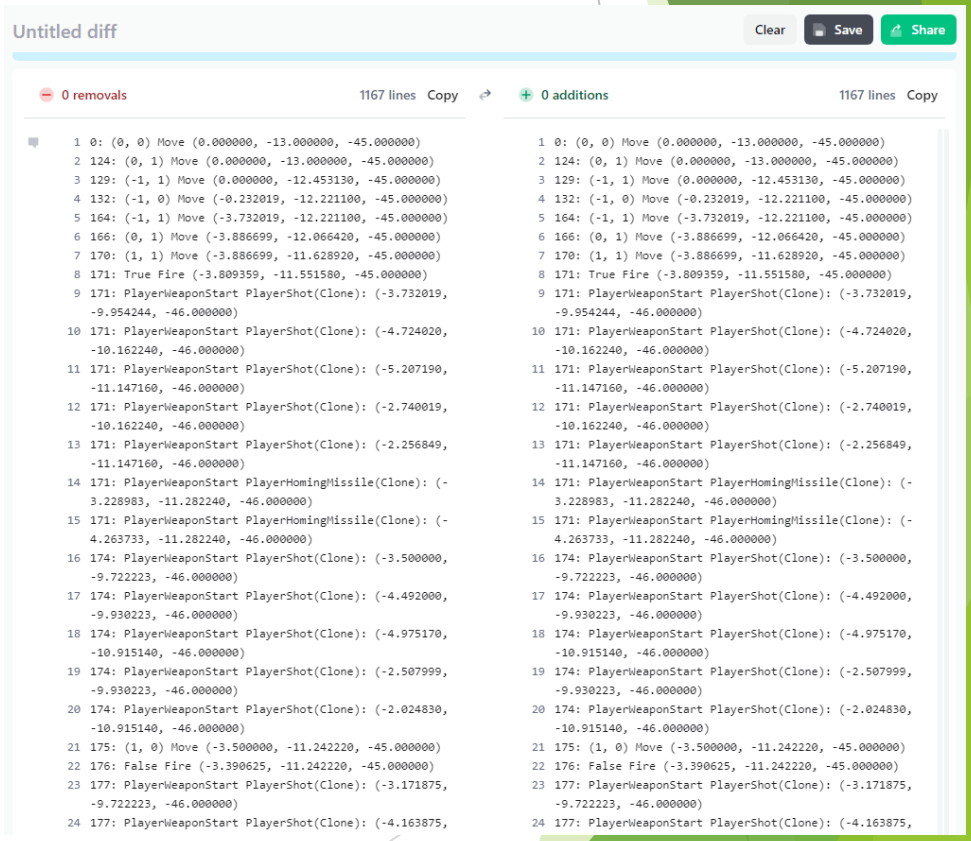
- 서로 충돌 체크가 필요한 TriggerBody끼리만 충돌 체크
(Unity의 Layer 개념)

- 꼭 필요한 곳에만 TriggerBody 사용.
(화면 바깥으로 나간 오브젝트에 대한 처리는 좌표 비교를 통해 처리)

구현 결과

- 동일한 리플레이 재생 시 반드시 동일한 결과가 발생하는 Deterministic한 리플레이 시스템 구현 성공

실제 플레이 화면 (좌), 리플레이 화면 (우)



리플레이를 두 번 재생했을 때, 게임 플레이 관련 로그를 비교해보면 100% 일치