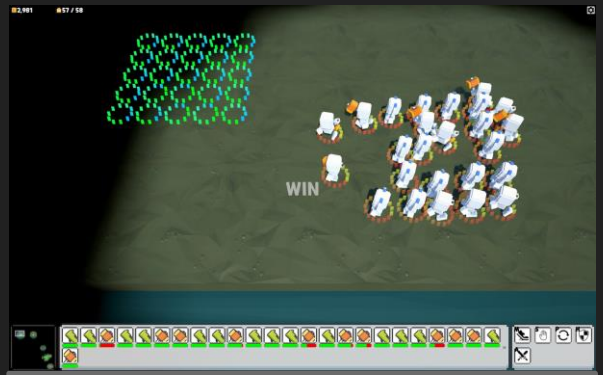
Survivor

이름 배성훈

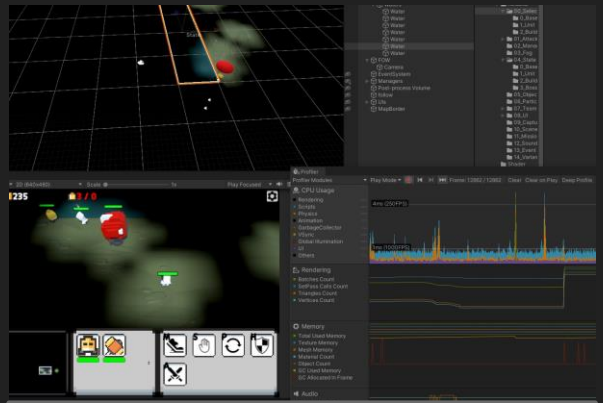
날짜 2025. 9. 15

목차

01. 게임 소개



게임 화면



제작 화면

제작 인원	1인
제작 시간	500시간
제작 목적	다수 유닛으로 최적화 경험
게임 장르	RTS

유니티
2021. 3.11f

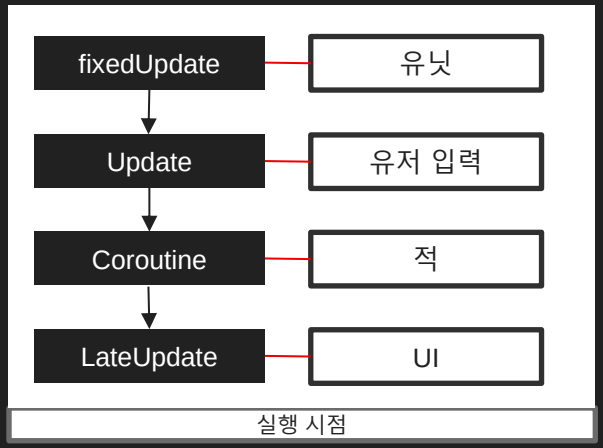
Visual Studio 2022
C# 9.0

포토샵

02. 게임 특징

RTS

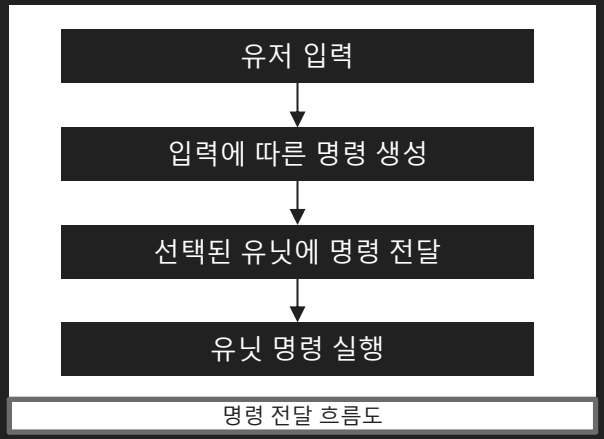
RTS 장르로 설계된 게임에서는 플레이어(유저와 적)가 각각 명령을 내리는 구조이고 유닛은 이러한 명령에 따라 움직이고 행동합니다.
→ 유저와 적이 지휘하고 유닛이 실행하게 설계했습니다.



실행 시점

유저(입력)	EventSystem 바로 다음 단계인 Update
유저(UI)	가장 최신 정보를 보여줘야 하므로 LateUpdate
적	일정 간격마다 실행되게 Coroutine
유닛	피격과 같이 물리 연산이 이뤄지므로 fixedUpdate

03. 주요 클래스



명령 전달 흐름도

```
<summary>
풀링 오브젝트! -> 유니티 pooling 기법을 !
</summary>
nity 스크립트(자산 참조 3개)|참조 42개
lic class PoolManager : MonoBehaviour


public static PoolManager instance;
[SerializeField] private PoolingData[] da
private Transform[] parents;
private Stack<GameObject>[] usedPrefabs;
```

풀은 스택으로 구현

PlayerManager – 유저와 유닛 사이를 연결하는 클래스입니다. 유저 입력 처리와 선택된 유닛에 명령 전달이 주 역할입니다.

UIManager – Unity UI의 변화를 일괄적으로 관리하는 클래스입니다. 변화 감지를 받고, `LateUpdate`에서 UI를 갱신합니다.

GameManager – 게임의 승패를 관리하는 클래스입니다.

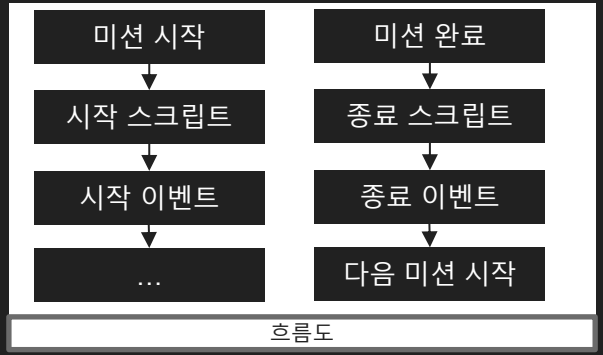
PoolManager – 오브젝트 풀링 데이터를 관리하는 클래스입니다. 타입별로 `Dictionary`에서 풀을 판별하고, 스택에서 꺼내 재사용하는 방식으로 구현했습니다

ActionManager – 게임 내 움직이는 오브젝트를 일괄 처리하는 클래스입니다. 유닛, 건물뿐만 아니라 미사일 등도 포함됩니다.

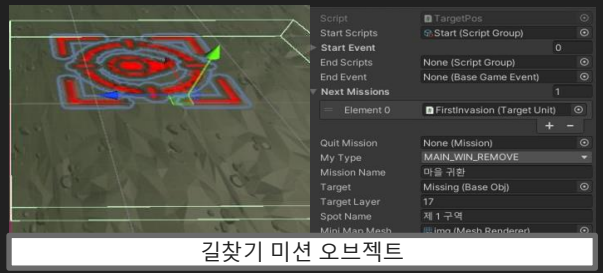
04. 미션

```
참조 2개
public bool IsMain { get { return (myType & MISSION.MAIN) != MISSION.NONE
참조 0개
public bool IsSub { get { return (myType & MISSION.SUB) != MISSION.NONE;
참조 1개
public bool IsHidden { get { return (myType & MISSION.HIDDEN) != MISSION.
참조 3개
```

미션 타입 정의



흐름도



길찾기 미션 오브젝트

숨겨진 미션 중 완료하면 게임이 강제 패배하는 미션 기믹이 존재하거나, 서브 미션 중 반복 가능한 미션이 존재합니다.

→ 여러 상태가 중첩 가능하므로 비트마스크로 이벤트 상태를 표현하고 비트 연산으로 타입 확인했습니다.

특정 지정 도달과 같은 미션이 존재하고, 지정 위치를 유니티 엔진으로 직접 확인하면서 조절하는게 좋다고 판단했습니다.

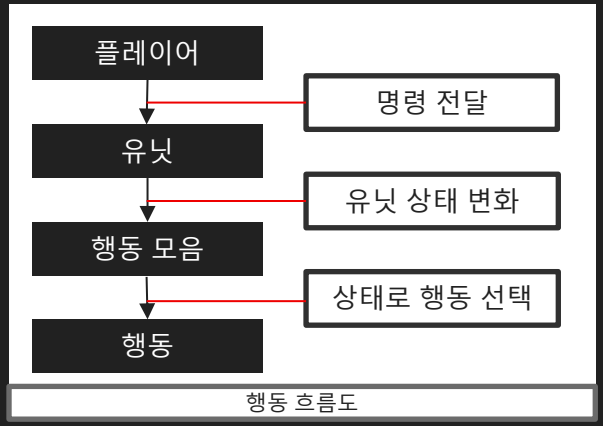
→ 미션은 MonoBehaviour를 상속해 컴포넌트로 설정합니다.

미션 중 깨면 끝나는 것들이 존재하지만 연계 미션도 고려했습니다.

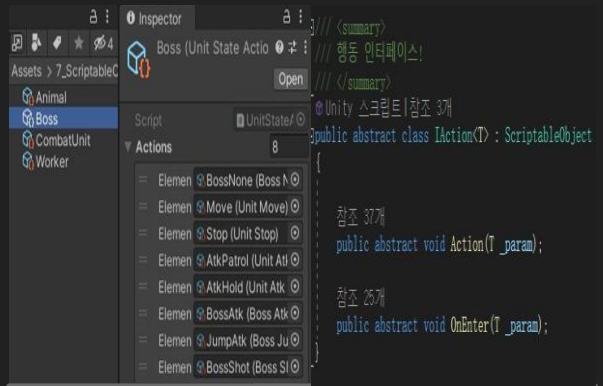
→ 단방향 연결리스트로 설계해 다음 미션으로 나아갈 수 있게 설정했습니다.

미션 이벤트는 미션 완료에 필요한 조건이나 추가적인 조건들을 준비 / 정리하는 역할을 합니다.

05. 유닛의 상태



행동 흐름도



행동 모음 / 행동

유닛 구현에는 크게 Behavior Tree와 FSM 방법으로 구현 방법이 존재합니다. 장르가 RTS 이므로 많은 유닛들이 존재할 수 있습니다.

→ 비교적 연산이 적은 FSM 알고리즘으로 유닛을 정의했습니다. 명령에 따라 행동이 변하며, 대부분 상태는 NONE으로 복귀하게 설계했습니다.

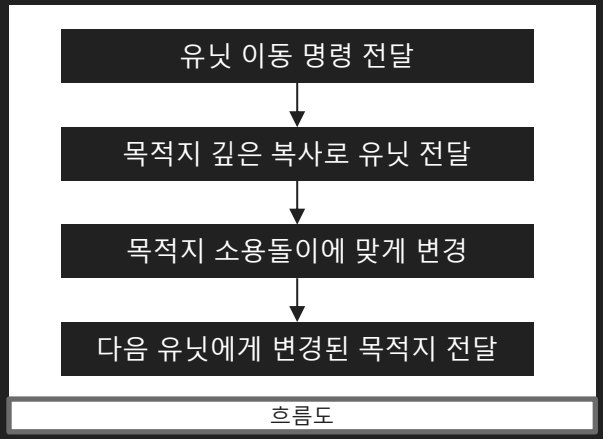
서포터 유닛처럼 이동은 가능한데 공격은 불가능한 유닛 방어 타워처럼 공격은 가능한데 이동이 불가능한 유닛, 이동과 공격이 모두 가능한 유닛 등 다양한 유닛들이 존재합니다.

행동이 유닛 스크립트 안에 정의되는 경우 새로운 타입의 유닛이 만들어지면 매번 새롭게 행동들을 정의해야 합니다.

→ 이를 개선하기 위해 유닛과 행동을 분리했습니다. 행동은 유닛 정보를 전달하면 행동이 수행했습니다.

또한, 유닛의 행동을 모아놓은 클래스 참조를 변경하여 인게임에서도 다양한 상태가 가능하도록 구현했습니다.

06. 유닛의 이동



흐름도

유닛의 이동은 NavMesh를 이용해 최단 경로를 따라가도록 구현했습니다. 목적지와의 거리가 0.5 이하이면 이동을 중지하도록 로직을 작성했습니다.

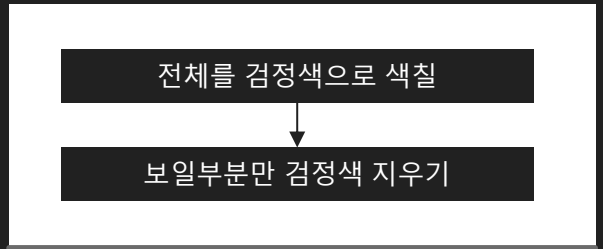
1마리 이동에서는 문제가 없으나, 다수의 유닛이 이동할 경우 모두 같은 좌표로 이동하려고 대부분이 도착했음에도 목적지 거리로 가기 위해 밀쳐내는 현상이 존재합니다.

한점에 모이는 문제 / 소용돌이 순서

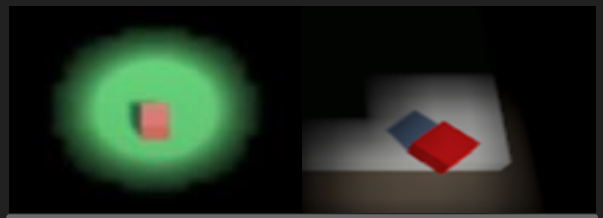
도착지 주변에 원형으로 유닛을 배치하여 최소 면적으로 이동하도록 고민했으나, 서로 다른 크기의 유닛이 존재하고 각도 연산 등 연산량이 많아지는 문제가 있습니다.

→ 단순 소용돌이 방식으로 배치하여 직사각형 형태를 유지하고, 유닛 크기에 따라 간격을 조절함으로써 문제를 해결했습니다.

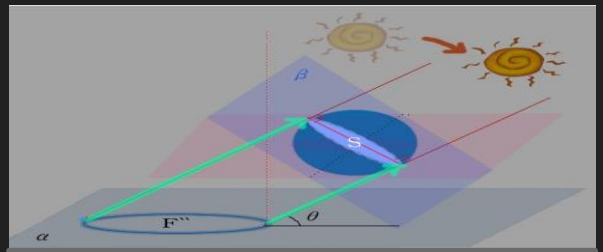
07. 전장의 안개



전장의 안개



카메라 + Layer / Projection + Renderer Texture



(1) 방법 문제 원인

유저에게 미지의 영역은 어둡게 했습니다.
→ 전장의 안개(Fog of War)를 적용했습니다.

전장의 안개 구현에는 두 가지 방법을 검토했습니다.
두 방법 모두 정사영을 이용해서 전장의 안개를 구현합니다.

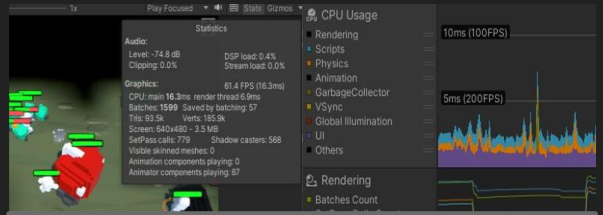
(1) 카메라 + Layer를 이용
(2) Projection과 Renderer Texture를 이용

첫 번째 방식은 플레인 오브젝트를 맵 위에 배치해 전체를 검정색으로 덮습니다. 이후 유닛 위치에 투명 원형 오브젝트를 두고, 카메라 Layer 설정을 통해 해당 영역만 보이도록 합니다.

→ 카메라가 기울어진 각도에서 투영되므로, 캐릭터 중심과 보이는 원형 시야가 어긋납니다. 이 때문에 캐릭터가 높낮이가 있는 지형을 이동할 때마다 추가 좌표 연산이 필요합니다.

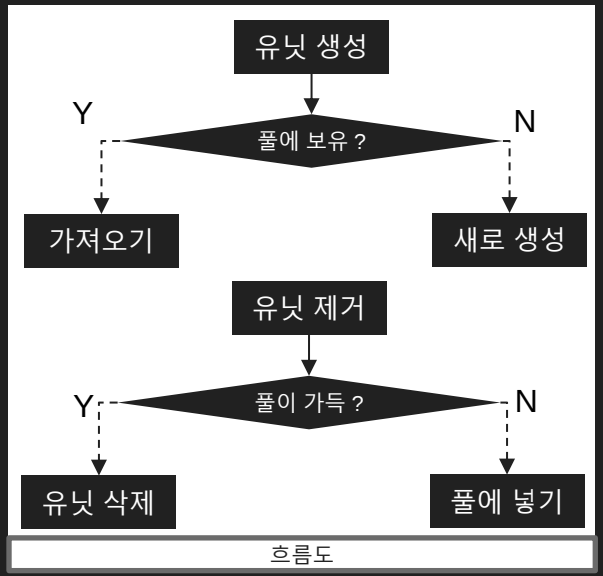
두 번째 방식은 Shader Projection으로 전체를 검정색으로 초기화하고, RenderTexture에 보이는 영역만 기록합니다. LateUpdate에서 이 텍스처를 셰이더에 곱해 최종적으로 보이는 부분만 표시합니다.
→ 추가 연산이 거의 필요하지 않아 최종적으로 채택했습니다.

08. 오브젝트 풀링



Stats 창 / 프로파일러

최적화 확인은 초기에는 유니티 엔진의 Game 창 Stats창의 FPS를 이용해서 비교했습니다.
이후 유니티 프로파일러를 사용해 others를 제외한 FPS를 확인했습니다.



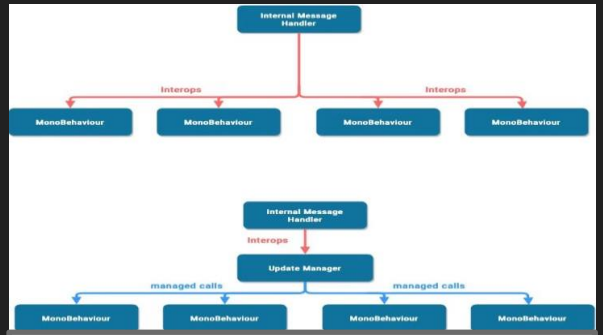
흐름도

오브젝트 풀링
RTS 장르 특성상 유닛과 미사일 등의 재사용이 많습니다.
→ 오브젝트 풀링 기법을 이용하여 파괴하지 않고 재사용함으로써 최적화를 시도했습니다.

초기에는 전체 미사일 중 비활성화된 것을 재활성화하는 방식으로 풀링을 구현했습니다.
→ 미사일 수가 적을 때는 문제가 없었으나, 대규모 전투에서는 프레임 저하가 발생했습니다.

원인은 미사일 발사 시 매번 모든 미사일을 조사하기 때문입니다.
→ 이후 풀을 별도로 관리하고, 스택으로 구현해 마지막에 사용했던 미사일을 재활용하는 방식으로 문제를 해결했습니다.

09. 구조적 개선



사용자 정의 업데이트 관리자 구축

유니티 공식 자료에 따르면, FixedUpdate 함수는 각 객체에 개별적으로 실행하는 것보다 실행할 객체를 자료구조에 모아 한 번에 처리하는 방식이 성능상 이점이 있다고 설명합니다.

따라서 행동을 수행하는 유닛들은 ActionManager에서 FixedUpdate를 일괄 실행하도록 설계했습니다.

→ 실제 테스트 결과, 약 10FPS 정도 성능 향상을 확인했습니다.

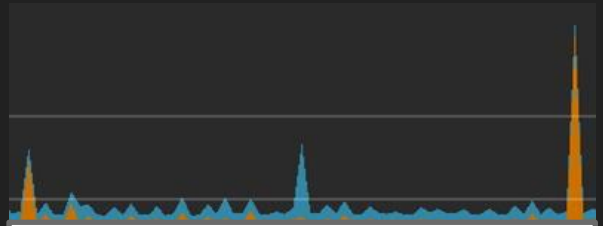
```
/// <summary> 범위안 타겟 찾기
public virtual void FindTarget(BaseObj _unit, bool _isChase, bool _isAlly = false)
{
    // 검사하는 유닛이 박스 콜라이더를 갖고 있어 hits는 최소 크기 1이 보장된다
    _unit.MyTurn++;
    if (_unit.MyTurn < chkTime) return;      // 일정 턴수마다 확인한다!
    _unit.MyTurn = 0;
```
일정 간격으로 탐색 연산

공격 가능한 유닛은 적 유닛이 근처에 있는지 탐색하기 위해 Physics.SphereCastAll을 사용하고, 레이어를 설정하여 일치하는 콜라이더를 확인합니다.

초기에는 매 FixedUpdate마다 탐색을 수행했으나, 공격 유닛 수가 많아지면서 Physics 연산량이 크게 증가하는 것을 확인했습니다.

이를 개선하기 위해 탐색 간격을 0.1초 단위로 설정하고 일정 시간마다 탐색하도록 변경했습니다.

→ 결과 물리 연산량 비율이 50% 이상 감소하는 것을 확인했습니다.


스파이크 구간이 적 탐색

10. 성능 확인 경험

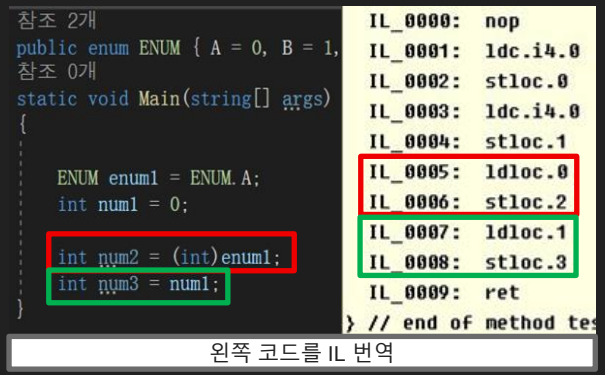
```
참조 2개
protected Dictionary<MY_STATE.GAMEOBJECT, int> MyActionNum
{

    get
    {

        if (myActionNum == null)
        {

            myActionNum = new Dictionary<MY_STATE.GAMEOBJECT, int>(actions.Length);
            for (int i = 0; i < actions.Length; i++)
            {

                myActionNum.Add((MY_STATE.GAMEOBJECT)i, i);
            }
        }

        return myActionNum;
    }
}
```

상태와 행동 잇는 Dictionary

유닛의 행동을 enum으로 표현하고, 행동들을 모아놓은 클래스에서 enum 값으로 행동을 찾아가는 방식을 고민했습니다.

고민은, enum을 형변환하여 배열에 행동을 보관하면 성능상 비용이 발생하지 않을까 하는 점이었습니다. 직접 확인할 수 없어서 혹시 비싼 연산일까봐 바로 적용하지 못했습니다.

대신 C#의 Dictionary는 원소를 찾는 연산이 O(1)로 빠르므로, enum과 상태 객체를 연결하는 방식으로 구현했습니다.



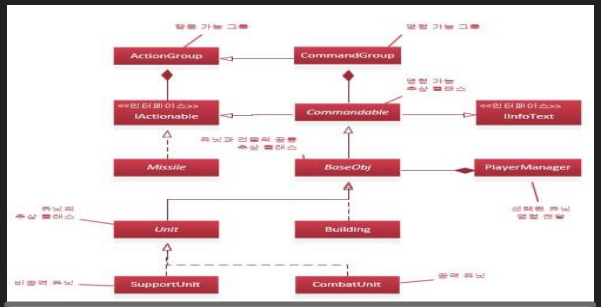
왼쪽 코드를 IL 번역

이후 C#의 지역 함수 동작이 C# 컴파일러에서 어떻게 하는지 확인하면서 enum 형변환 역시 같은 방법으로 접근했습니다.

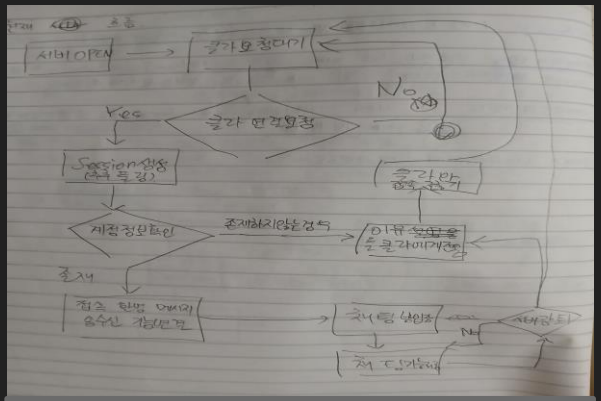
ILDASM으로 확인한 결과 형변환 연산 비용이 없습니다.

결론적으로, enum을 이용한 접근은 효율적이라는 판단을 내렸고, 이 과정에서 **직접 성능을 검증하고 안전하게 구조를 설계하는 경험**을 얻었습니다.

11. 느낀점



클래스 관계도



채팅 프로그램 진행 과정

클래스를 세분화하면 기능 확장은 용이하지만, 관련 클래스까지 함께 고려해야 하므로 연관 기능을 찾는 데 시간이 많이 걸리는 경험을 했습니다.

이를 통해 클래스 간 관계도를 기록하고 프로그램 진행 과정을 파악하는 것이 중요함을 깨달았습니다.

현재는 코드 작성 전에 클래스 관계를 설계하고 노트에 정리하는 습관을 유지하고 있습니다.

마무리

게임 영상

https://blog.naver.com/tryingpop/224008046530