

class ObjectPoolingManager 설계

By 남정웅

Index

0. 설계 목표
1. class ObjectPoolingManager
2. 인스펙터뷰에서 풀 편집
3. 게임 시작 후 다른 클래스에서 사용 할 때
4. 객체가 사라질 때
5. 활성화 시킬 객체가 없을 때
6. 리팩토링& 컴파일링

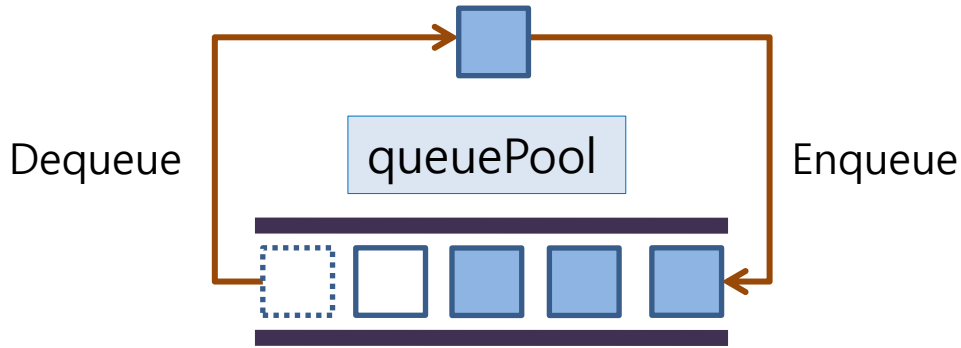
0. 설계 목표

- 1) 프로그래머가 아니더라도 유니티상에서 쉽게 사용 할 수 있게 설계
- 2) 하나의 `class`에서 풀들을 일괄 관리(인스펙터뷰에서 편집 가능)
- 3) 하나의 풀을 여러 클래스에서 사용 할 수 있게 설계
- 4) 게임시작전 유니티 에디터에서 미리 객체들을 생성가능
- 5) `Instantiate`와 같이 인스펙터뷰에서 프리팹을 할당하는 형식으로 사용
- 6) 풀에서 활성화 시킬 객체가 없는 경우 처리방법 선택가능

1. class ObjectPoolingManager

- 1) 풀링 할 객체들의 풀을 모두 이 클래스에서 관리(싱글톤)
- 2) 풀의 자료구조는 Queue이용
- 3) Dictionary를 활용하여 하나의 풀을 여러 클래스에서 사용 할 수 있게 설계
- 4) enum분기를 통해 풀에 있는 객체가 모두 활성화 중일 때 어떻게 처리할지 구현
- 5) 일반적인 Instantiate처럼 인스펙터뷰에서 프리팹을 할당해서 사용
- 6) 오브젝트들을 풀링 할 때 class IsReusing 컴포넌트를 추가하며 IsReusing 컴포넌트의 유무에 따라 객체가 사라 질 때 ObjectPoolingManager의 DisableOrDestroy() 함수를 통해 비활성화 시킬지 파괴 할지 결정

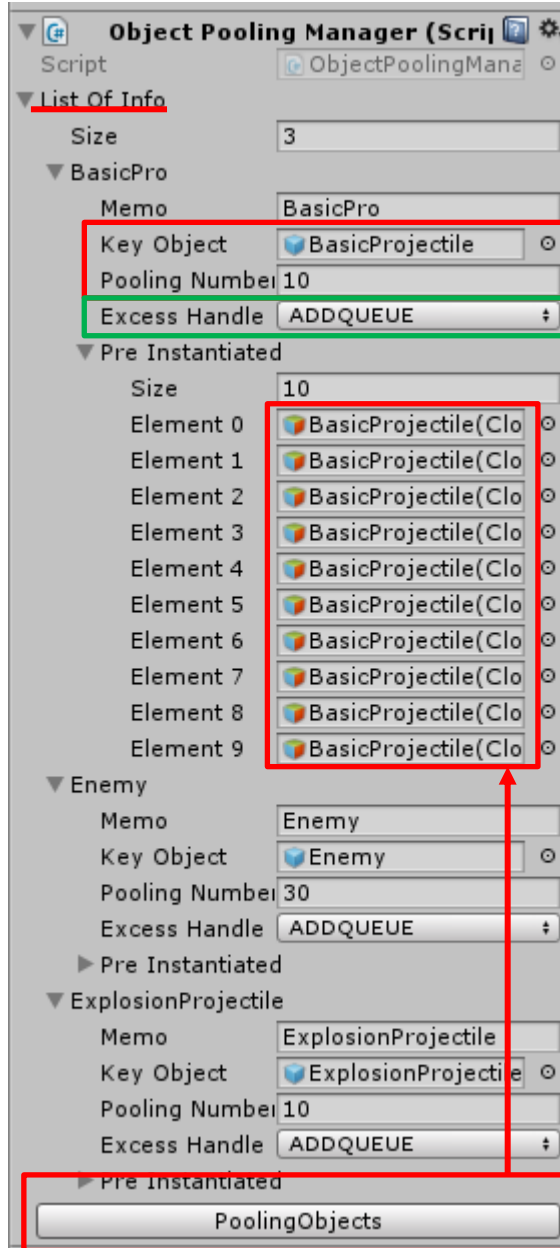
*풀을 Queue로 만드는 이유



Dequeue후 활성화 여부 확인
이미 활성화 되었다면 Dequeue후 반복
비활성화 되었다면 Enqueue후 종료

- 1) 먼저 활성화된 객체가 먼저 비활성화 되는 경우가 많다고 가정하고 Dequeue, Enqueue로 정렬 함
- 2) 풀의 크기 만큼 객체를 사용한 이후 다시 비활성화된 객체를 찾을 때 Queue의 이 전에 가장 먼저 활성화 된 첫 번째 요소가 먼저 사용이 끝나고 비활성화 되어 있을 가능성이 높음(최선의 경우)
- 3) 이 로써 풀에서 비활성화 된 객체를 탐색 하는 알고리즘에서 시간 복잡도를 분석 했을 때 최선의 경우 (빅오 표기법으로) $O(1)$ 의 실행시간을 기대 할 수 있음 (최악의 경우 $O(n)$)
- 5) 최선의 경우 Dequeue, Enqueue로 정렬 하지 않을 때에 비해 Dequeue, Enqueue 비용이 더 들어 가지만, 최악의 경우가 적어지므로 탐색시간의 차이를 줄일 수 있음

2. 인스펙터뷰에서 풀 편집



미리 만들어 놓고 싶은
오브젝트와 그 숫자

풀에 활성화 시킬 객체가 없을 경우 처리방법 선택

List<InfoOfPool> listOfInfo

InfoOfPool

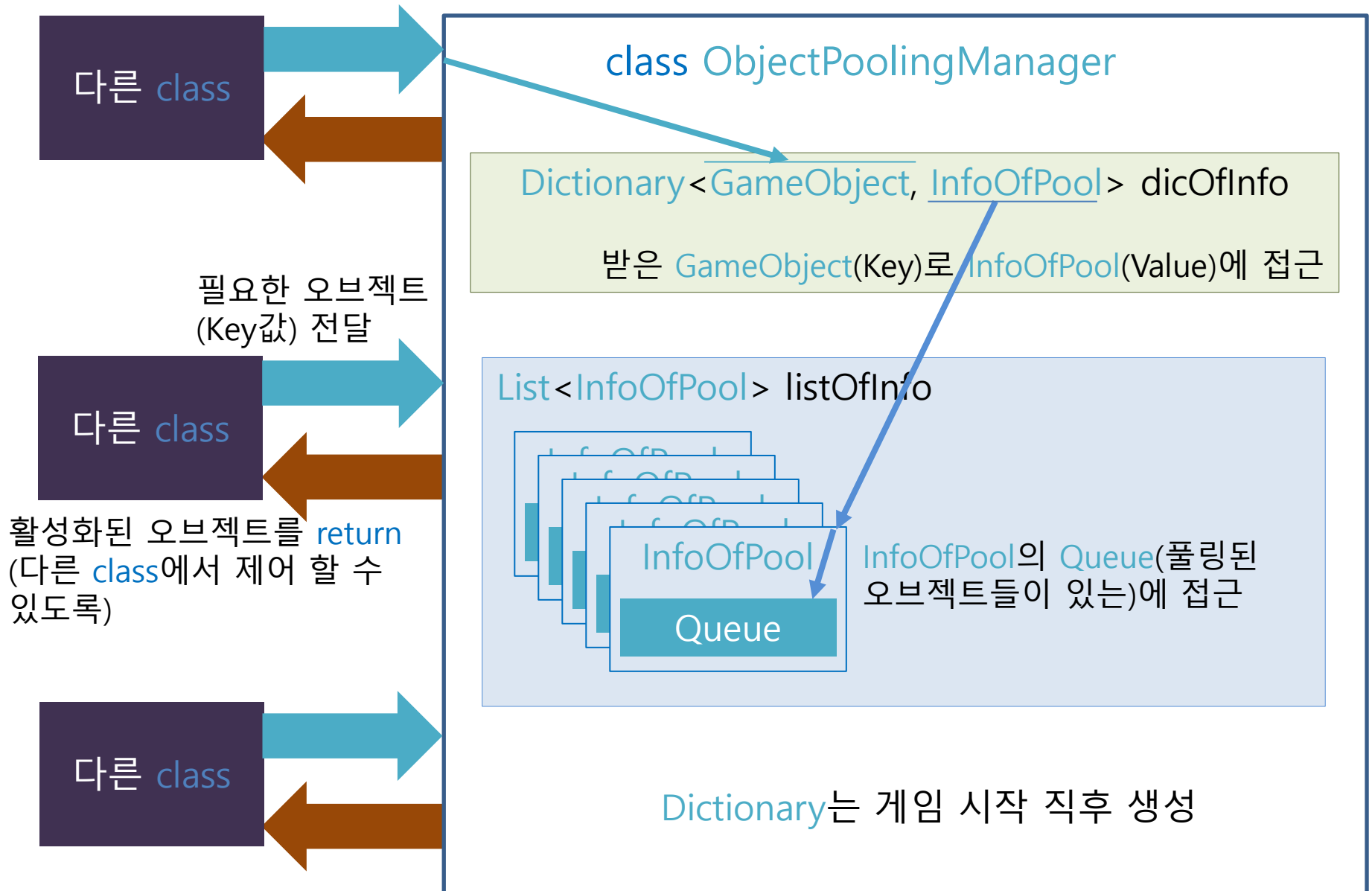
Queue

(인스펙터뷰에는 표시되지 않지만
InfoOfPool에는 Queue도 선언되어 있음)

생성된 인스턴스들은 게임 시작 후
Queue로 된 풀에 Enqueue

버튼을 누르면 KeyObject와 PoolingNumber의 값을
토대로 미리 풀에 넣을 인스턴스들을 생성함

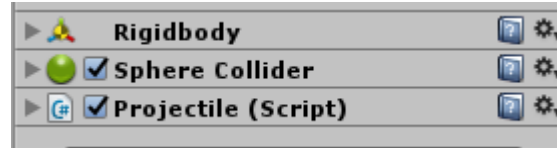
3. 게임 시작 후 다른 클래스에서 사용 할 때



4. 객체가 사라질 때

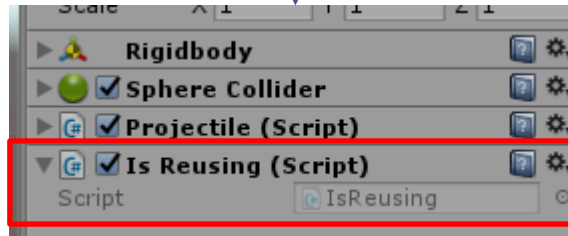
그 자체로는 아무런 기능이 없는, `class ObjectPoolingManager`에서 풀링 된 객체에 추가 시키며 재사용 가능 여부를 판단하기 위한 컴포넌트용 `class`

프리팸



오브젝트가 풀링 될 때
(ObjectPooling(...)함수 실행 시)
자동으로 컴포넌트가 추가됨

`class ObjectPoolingManager`
를 통해 풀링 되었을 때



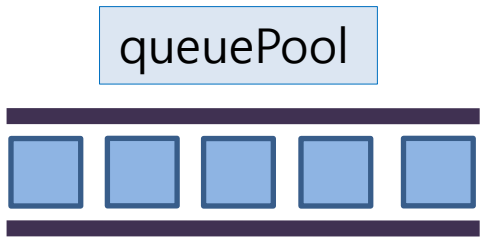
이 객체가 사라 질 때 `class ObjectPoolingManager`의 `DisableOrDestroy` 함수를 실행
`DisableOrDestroy`에서 이 컴포넌트의 유무로 비활성화 시킬지 파괴시킬지 결정

이 방법을 통해 풀을 이용한 방법이 아니라도
일반적인 `Instantiate`을 통한 방법도 사용가능

5. 활성화 시킬 객체가 없을 때

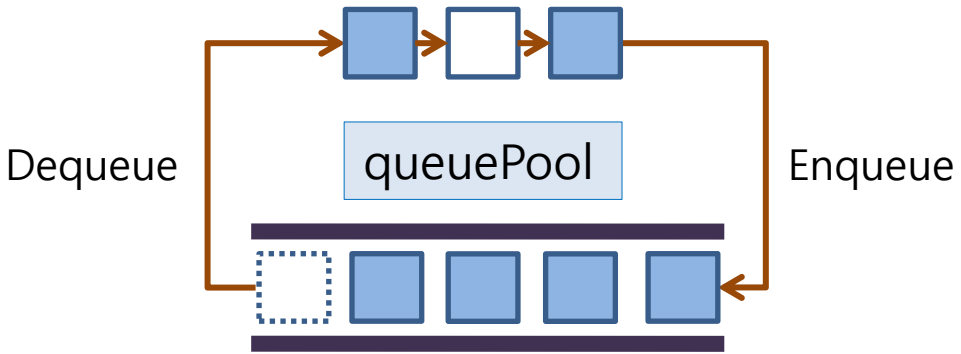
풀에 활성화시킬 객체가 없을 때 처리방법

1) DONOTMAKE



아무런 행동을 하지 않는다

2) DEACTIVEFIRST

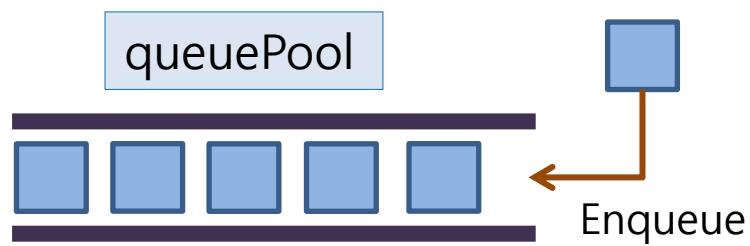


가장 먼저 활성화 되었던 객체(Queue의 가장 앞)를 비활성화 한 후 다시 활성화 (Transform은 다시 초기화)

인스펙터뷰에서 ExcessHandle 선택가능

```
//큐에있는 모든 객체가 활성화되어
//더 이상 활성화시킬 객체가 없을때 어떻게 처리할지 결정
public enum ExcessHandle
{
    //아무런 행동을 하지 않는다
    DONOTMAKE,
    //가장 먼저 활성화됐었던(Queue의 가장 앞) 객체를
    //비활성화 한 후 다시 활성화
    DEACTIVEFIRST,
    //새로운 객체를 생성하여 풀에 추가
    ADDQUEUE,
    //새로운 객체를 생성 하되 풀에 추가 하지 않음
    DONOTREUSE
}
```

3) ADDQUEUE

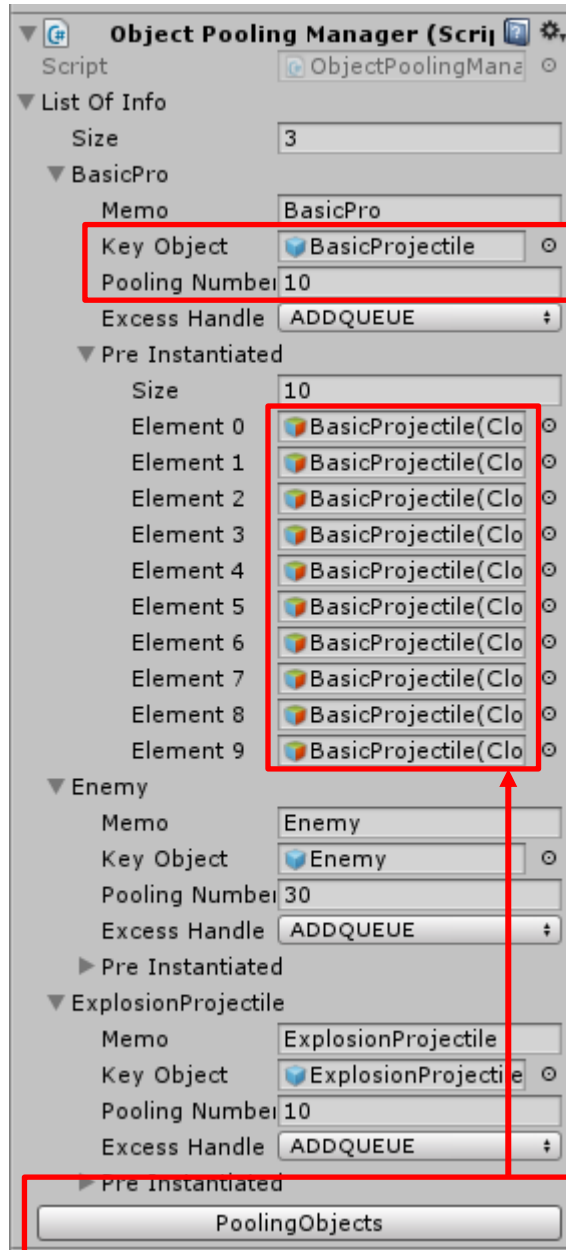


객체를 새로 Instantiate한 후 큐에 추가 (IsReusing컴포넌트가 있는 상태)

4) DONOTREUSE

Instantiate만 하고 큐에 추가하지 않음 ((IsReusing컴포넌트가 없는 상태)

6. 리펙토링& 컴파일링

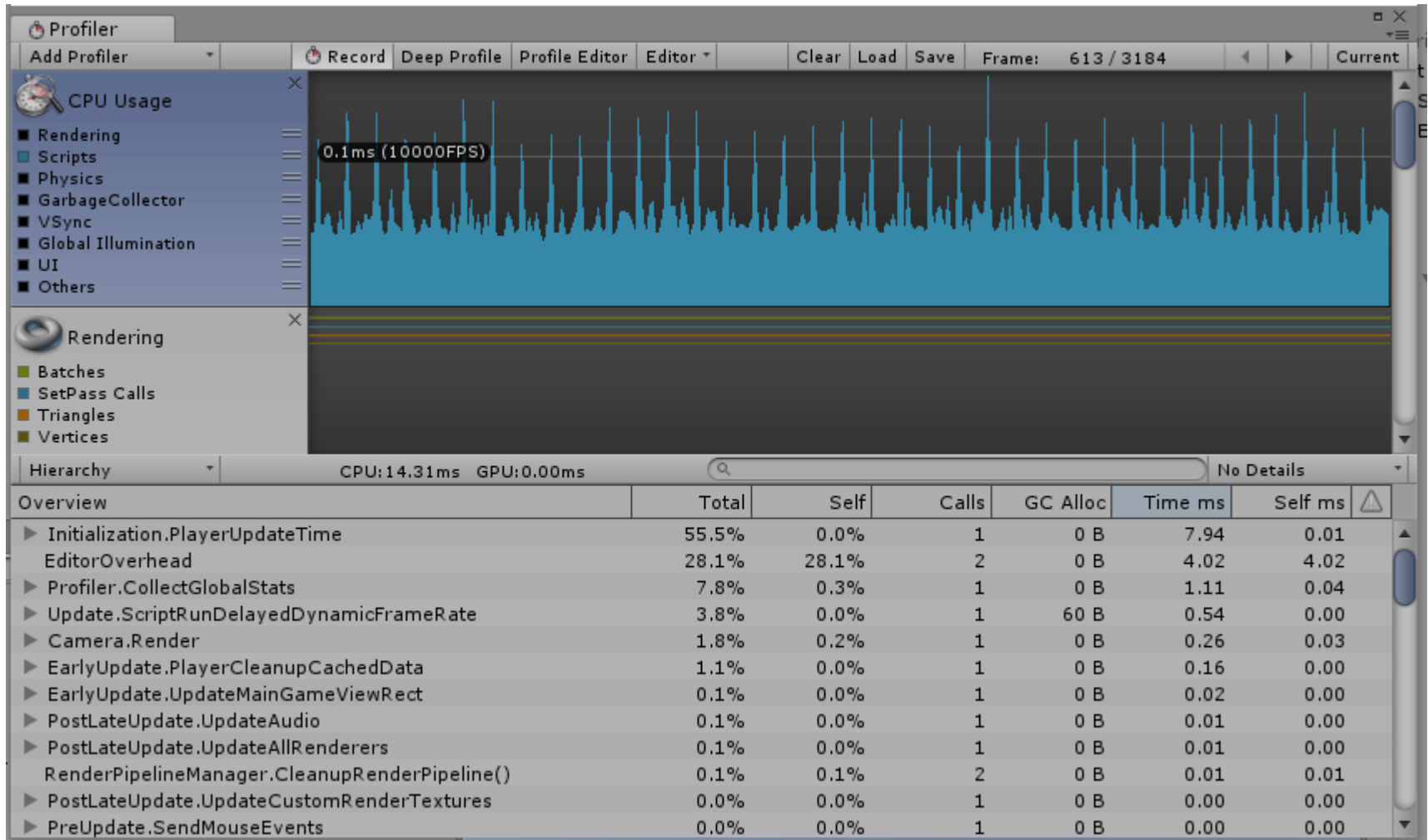


미리 풀을 생성하는 경우 프리팹을 수정 했을 때 이미 생성된 인스턴스들은 수정되지 않기 때문에 수정 할 때마다 PoolingObjects버튼 를 눌러 줘야 함

그러므로 버튼을 누르는것 대신 `class ObjectPoolingManager`의 `Awake()`에서 실행 프리팹을 수정해도 버튼을 눌러 줄 필요가 없음

게임 시작 시 인스턴스들을 생성 할 때 드는 비용은 게임 시작시간을 늦추는 것으로 해결

1000개의 풀을 생성후 0.1초 마다 풀의 객체를 활성화 시키고 활성화된 객체는 1초마다 비활성화 시킴



객체를 활성화 시키는 0.1초 마다 스크립트 실행시간이 최대 0.15ms이하로 스크립트에 의한 퍼포먼스의 문제는 발생하지 않음