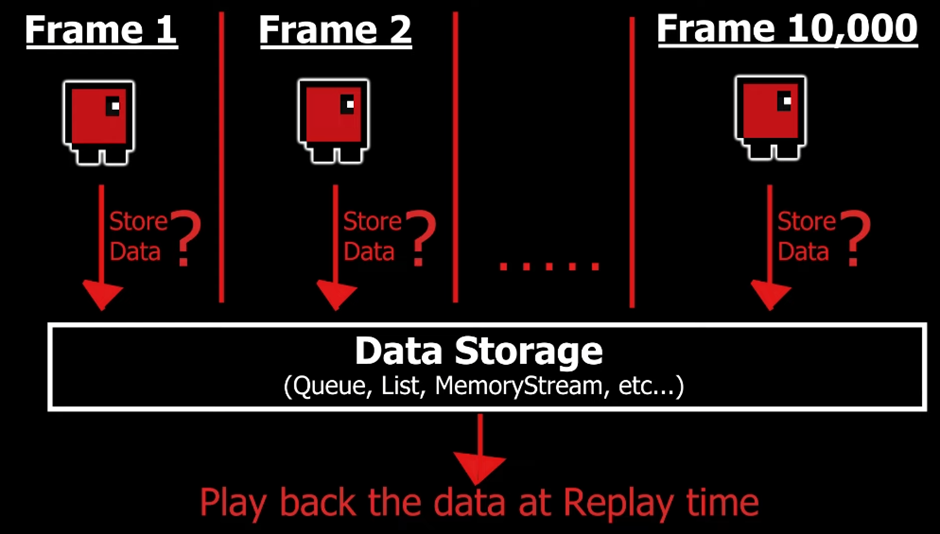
**본 글은 엔픽셀 인턴십 중 작성한 문서로서 회사의 허락을 받아 공개합니다.**

## **Replay System의 전반적인 개요**



1. 각 프레임에서 데이터를 저장
2. 저장된 데이터를 Replay Time에 차례대로 실행

여기서 어떤 데이터를 저장할 것인가??에 따라 Replay 구현 방식이 크게 두가지로 나뉩니다.

## **Input Based Replay**

Player의 Input (Command)를 저장하는 방식입니다.

상대적으로 저장되는 데이터의 양이 적은 방식이나, 모든 Input의 결과가 Deterministic해야만 사용할 수 있습니다.

-> 모든 Input에 대해 같은 결과가 보장되어야 함. (안타깝게도 Unity의 Physics-System은 Non-Deterministic합니다. Input Based Replay방식을 구현하고자 하시는 분들은 직접 Physics-System을 구성하여 적용해야 할 것입니다.)

## **State Based Replay**

Player를 포함하여 Replay할 모든 가변 객체의 State를 저장하는 방식입니다.

상대적으로 저장되는 데이터의 양이 많은 방법이며, Non-Deterministic Engine에서도 사용할 수 있는 방식입니다. 현재 문서에서는 비교적 단순한 State Based Replay방식을 위주로 설명 할 것입니다.

## **Replay System을 구성하는 기본 Class**

1. Recorder : 데이터를 저장하고 재생 (캠코더)
2. Recording : ReplayObject와 ReplayData의 묶음 (동영상)
3. ReplayData : Replay할 데이터의 묶음 (행동, 상태)
4. ReplayObject : Replay할 오브젝트. Replay에 사용되는 오브젝트는  
    원래 오브젝트의 Prefab보다 간소화된 오브젝트를 사용합니다. (주체)  
     
    여러개의 ReplayObject와 ReplayData가 필요한 경우, ReplayObject,  
    ReplayData의 하위 클래스를 파생하여 관리하는 것이 보다 간편합니다.  
     
    또한 데이터의 저장은 오브젝트의 상태 변경이 처리된 이후에 관리하는 것이 좋으며, 따라서 유니티의 모든 Update관련 이벤트 함수 중 가장 늦게 호출되는 LateUpdate에서 처리하는 것이 가장 바람직합니다.

## **하나의 오브젝트를 저장하는 방법**

하나의 플레이어 캐릭터를 기준으로 transform.position 값만을 매 프레임 저장하고 재생하는 리플레이 시스템 구현 방식은 아래와 같습니다. RecordTarget은 PlayerController와 같이, 저장할 리플레이 데이터를 주로 가지고 있는 스크립트가 대상이 됩니다. 매 프레임 데이터를 저장하고 데이터를 저장된 순서대로 사용하기 때문에 FIFO 자료구조인 Queue를 사용합니다. System.IO에서 제공하는 MemoryStream을 사용하는 방법도 존재하지만, 현재 문서에서는 다루지 않겠습니다. MemoryStream을 사용한 구현이 궁금하신 분들은 문서 하단부의 참고 자료를 이용해주세요.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Recorder : MonoBehaviour

{

[SerializeField] private GameObject replayObjectPrefab;

//리플레이 정보를 저장할 Queue

public Queue<ReplayData> recordingQueue { get; private set; }

private Recording recording;

//리플레이 재생 여부 확인

private bool isDoingReplay = false;

private void Awake()

{

recordingQueue = new Queue<ReplayData>();

}

//리플레이 재생

private void Update()

{

if (!isDoingReplay)

{

return;

}

bool hasMoreFrames = recording.PlayNextFrame();

if (!hasMoreFrames)

{

RestartReplay();

}

}

//리플레이를 시작하는 이벤트가 발생 시

private void OnReplayStart()

{

StartReplay();

}

//리플레이를 종료하는 이벤트가 발생 시

private void OnReplayFinish()

{

Reset();

}

//매 프레임 ReplayData를 저장

public void RecordReplayFrame(ReplayData data)

{

recordingQueue.Enqueue(data);

}

//기록된 Replay 시작

private void StartReplay()

{

isDoingReplay = true;

recording = new Recording(recordingQueue);

recordingQueue.Clear();

recording.InstantiateReplayObject(replayObjectPrefab);

}

//리플레이 종료 후 재시작

private void RestartReplay()

{

isDoingReplay = true;

recording.RestartFromBeginning();

}

//저장된 리플레이 데이터 초기화

private void Reset()

{

isDoingReplay = false;

recordingQueue.Clear();

recording.DestroyReplayObjectIfExists();

recording = null;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Recording

{

public ReplayObject replayObject { get; private set; }

//리플레이 재생에 사용될 Queue

private Queue<ReplayData> replayQueue;

//리플레이 재시작에 사용될 Queue

private Queue<ReplayData> originalQueue;

public Recording(Queue<ReplayData> recordingQueue)

{

this.replayQueue = new Queue<ReplayData>(recordingQueue);

this.originalQueue = new Queue<ReplayData>(recordingQueue);

}

//리플레이 재시작

public void RestartFromBeginning()

{

this.replayQueue = new Queue<ReplayData>(originalQueue);

}

//리플레이 재생

public bool PlayNextFrame()

{

if (replayObject == null)

{

Debug.LogError("리플레이 할 오브젝트가 존재하지 않습니다.");

}

bool hasMoreFrames = false;

if (replayQueue.Count != 0)

{

ReplayData data = replayQueue.Dequeue();

replayObject.SetDataForFrame(data);

hasMoreFrames = true;

}

return hasMoreFrames;

}

//리플레이 오브젝트 생성

public void InstantiateReplayObject(GameObject replayObjectPrefab)

{

if (replayQueue.Count != 0)

{

ReplayData startingData = replayQueue.Peek();

this.replayObject = Object.Instantiate(replayObjectPrefab, startingData.position, Quaternion.identity).GetComponent<ReplayObject>();

}

}

//리플레이 오브젝트 파괴

public void DestroyReplayObjectIfExists()

{

if (replayObject != null)

{

Object.Destroy(replayObject.gameObject);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class ReplayData

{

//리플레이에 사용할 변수...

public Vector3 position { get; private set; }

public ReplayData(Vector3 position)

{

this.position = position;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class ReplayObject : MonoBehaviour

{

//리플레이 오브젝트의 상태를 매 프레임 갱신

public void SetDataForFrame(ReplayData data)

{

this.transform.position = data.position;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class RecordTarget : MonoBehaviour

{

//현재 오브젝트의 데이터를 기록하기 위해 Recorder컴포넌트 추가

private Recorder \_recorder;

private void Awake()

{

\_recorder = GetComponent<Recorder>();

}

private void LateUpdate()

{

//매 프레임 ReplayData를 생성하고 recorder에 저장

ReplayData data = new ReplayData(this.transform.position);

\_recorder.RecordReplayFrame(data);

}

}

## **다수의 오브젝트를 저장하는 방법**

위의 코드를 토대로 여러개의 오브젝트에서 리플레이 데이터를 저장하고 재생하는 시스템을 만들 수 있습니다. 먼저 다양한 형태의 ReplayData를 저장하기 위해 ReplayData를 abstract class로 변경 후 파생 클래스를 생성하여 ReplayData를 저장합니다.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public abstract class ReplayData

{

//리플레이에 사용할 변수...

public Vector3 position;

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PlayerReplayData : ReplayData

{

public PlayerReplayData(Vector3 position)

{

this.position = position;

}

}

또한 ReplayObject의 다양화를 위해 ReplayObject역시 abstract class로 변경 후 파생클래스를 생성합니다.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public abstract class ReplayObject : MonoBehaviour

{

//리플레이 오브젝트의 상태를 매 프레임 갱신

public abstract void SetDataForFrame(ReplayData data);

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PlayerReplayObject : ReplayObject

{

//플레이어 캐릭터 리플레이에 필요한 컴포넌트...

private void Awake()

{

//컴포넌트 초기화

}

public override void SetDataForFrame(ReplayData data)

{

PlayerReplayData playerData = (PlayerReplayData)data;

//플레이어 캐릭터 컴포넌트 데이터 업데이트 ex) position, animation...

}

}

## **Replay System의 단순 구현 코드**

상기 내용들을 정리하여 단순한 Replay System을 구현하면 아래와 같습니다. Recorder와 Recording클래스는 거의 유사하며, 현재 코드는 PlayerCharacter를 기준으로 작성되었습니다. 다른 오브젝트를 저장하고 싶다면 해당 오브젝트에 필요한 ObjectReplayData, ObjectReplayOjbect 클래스 작성 후 RecordTarget에 Recorder 컴포넌트 추가, LateUpdate에 작성한 자료형의 data를 사용하면 됩니다.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Recorder : MonoBehaviour

{

[SerializeField] private GameObject replayObjectPrefab;

public Queue<ReplayData> recordingQueue { get; private set; }

private Recording recording;

private bool isDoingReplay = false;

private void Awake()

{

recordingQueue = new Queue<ReplayData>();

}

private void Update()

{

if (isDoingReplay)

{

return;

}

if (recording.HasMoreFrames())

{

recording.PlayNextFrame();

}

else

{

RestartReplay();

}

}

public void RecordReplayFrame(ReplayData data)

{

recordingQueue.Enqueue(data);

}

private void StartReplay()

{

isDoingReplay = true;

recording = new Recording(recordingQueue);

recordingQueue.Clear();

recording.InstantiateReplayObject(replayObjectPrefab);

}

private void RestartReplay()

{

isDoingReplay = true;

recording.RestartFromBeginning();

}

public void ResetRecorder()

{

isDoingReplay = false;

recordingQueue.Clear();

recording.DestroyReplayObjectIfExists();

recording = null;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Recording

{

public ReplayObject replayObject { get; private set; }

private Queue<ReplayData> originalQueue;

private Queue<ReplayData> replayQueue;

public Recording(Queue<ReplayData> recordingQueue)

{

this.originalQueue = new Queue<ReplayData>(recordingQueue);

this.replayQueue = new Queue<ReplayData>(recordingQueue);

}

public void RestartFromBeginning()

{

this.replayQueue = new Queue<ReplayData>(originalQueue);

}

public void PlayNextFrame()

{

if (replayObject == null)

{

Debug.LogError("리플레이할 오브젝트가 존재하지 않습니다.");

}

if(replayQueue.Count != 0)

{

ReplayData data = replayQueue.Dequeue();

replayObject.SetDataForFrame(data);

}

}

//더 재생할 프레임이 남아있는지 확인

public bool HasMoreFrames()

{

if (replayQueue.Count == 0)

{

return false;

}

return true;

}

public void InstantiateReplayObject(GameObject replayObjectPrefab)

{

if (replayQueue.Count != 0)

{

ReplayData startingData = replayQueue.Peek();

this.replayObject = Object.Instantiate(replayObjectPrefab, startingData.position, Quaternion.identity).GetComponent<ReplayObject>();

}

}

public void DestroyReplayObjectIfExists()

{

if (replayObject != null)

{

Object.Destroy(replayObject.gameObject);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PlayerReplayData : ReplayData

{

//플레이어 캐릭터 리플레이에 필요한 변수...

public PlayerReplayData(Vector3 position)

{

this.position = position;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PlayerReplayObject : ReplayObject

{

//플레이어 캐릭터 리플레이에 필요한 컴포넌트...

private Animator animator;

private void Start()

{

animator = GetComponentInChildren<Animator>();

}

public override void SetDataForFrame(ReplayData data)

{

PlayerReplayData playerData = (PlayerReplayData)data;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class RecordTarget : MonoBehaviour

{

private Recorder recorder;

private void Start()

{

recorder = GetComponent<Recorder>();

}

//오브젝트 로직... (플레이어 캐릭터라면 캐릭터 로직)

private void LateUpdate()

{

ReplayData data = new PlayerReplayData(this.transform.position);

recorder.RecordReplayFrame(data);

}

}

## **Replay System 최적화 방법**

· ReplayData 간소화

Replay System의 최적화 방법은 '어떤' 데이터를 '어떻게' 저장할 것인가 에 주로 초점이 맞춰집니다. State Based Replay의 경우 매 프레임 리플레이에 필요한 데이터를 모두 저장하기에, 저장할 데이터의 자료형 또한 최적화에 많은 영향을 미치게 됩니다.

o Ex) 60프레임 게임을 5분 저장하는 경우, int형 변수 하나를 저장하는데 필요한 메모리는 60 \* 300 \* 4byte = 72,000byte -> 70Kbyte 이며, 이 정보를 bool형 변수로 처리할 수 있다면 필요 메모리는 60 \* 300 \* 1byte = 18,000byte -> 17.6Kbyte로 약 52.4Kbyte의 메모리 공간을 절약할 수 있습니다.

o Ex) 어떤 정보를 저장하는데 필요한 자료형이 int이지만 해당 정보를 사용하는 빈도가 매우 낮은 경우, 해당 정보를 계속해서 저장하는 것 보다 bool변수로 trigger설정을 하여, 해당 정보가 사용될 때만 bool변수를 true처리하여 저장하는 방법이 있습니다. 이렇게 설정하면 bool변수가 false인 경우 해당 정보를 저장하지 않게 되어 불필요한 정보 저장을 막을 수 있습니다.

o Ex) 의도적으로 저장할 프레임 주기를 낮추는 방법이 있습니다. 매 프레임 저장하던 정보를 의도적으로 2프레임 단위로 저장하는 경우, 저장할 데이터의 양이 50% 감소하게 되어 메모리 관점에서 큰 이득을 가져올 수 있습니다. 다만 저장할 프레임 단위를 과하게 설정할 경우, 리플레이 도중 끊기는 현상이 발생할 수 있으니 본인 게임에 최적화된 프레임 단위를 찾는 것이 중요합니다.

이처럼 Replay System을 설계 할 때는 저장할 정보가 반드시 필요한 정보인지, 필요한 정보라면 그 정보를 처리하는데 필요한 최소 자료형이 무엇인지, 해당 정보들을 어떻게 저장할 것인지 설정하는 것이 중요합니다.

· ReplayObject 경량화

리플레이에 사용되는 Object는 단순히 저장된 데이터를 받아서 재생할 뿐, 그 외의 어떠한 기능도 필요하지 않다는 사실에 입각한 최적화 방법입니다. 예를 들어 플레이어 캐릭터를 리플레이 하는 경우, 리플레이 오브젝트는 플레이어의 입력 처리는 물론 충돌 처리, trigger처리 마저 할 필요가 없을 수 있습니다. 이는 저장할 리플레이 정보에 이미 처리된 경우가 대부분 일 것이며, 따라서 ReplayObject에는 해당 기능을 처리할 컴포넌트가 존재하지 않아도 됩니다. 오브젝트 재생에 필요한 최소한의 컴포넌트만 남겨두고 그 외의 컴포넌트를 제거하여 ReplayObject를 경량화 할 수 있습니다.

## **참고자료**

<https://www.youtube.com/watch?v=ilOQstDnX2I>

<https://www.raywenderlich.com/7728186-creating-a-replay-system-in-unity>

<https://www.gamedeveloper.com/programming/implementing-a-replay-system-in-unity-and-how-i-d-do-it-differently-next-time>

<https://docs.microsoft.com/ko-kr/dotnet/api/system.io.memorystream?view=net-6.0>

<https://stackoverflow.com/questions/5413801/queuebyte-vs-stream>