Rewind System - Documentation

Table of Contents

1. Introduction

The **Rewind System** is a robust framework for **recording** and **rewinding** the state of Unity objects over time. Its design focuses on:

- **Modularity**: Supporting various components like Transforms, Rigidbodies, Animators, Particle Systems, and more.

- **Extensibility**: By leveraging an abstract base class (RewindableStructBase), you can create new rewinders for custom components or additional data.
- **Performance**: Uses a fixed-capacity **ring buffer** to avoid unbounded memory usage.
- **Ease of Integration**: Minimal friction when adding to existing Unity projects.

2. High-Level Architecture

2.1 Core Concepts and Flow

1. **Recording**: The system periodically captures snapshots of an object's state. Each snapshot is stored in a ring buffer (StructRingBuffer<T>).

2. **Buffer**: Since the buffer has a fixed capacity, once it's full, older snapshots are removed or overwritten (depending on implementation).

3. **Rewinding**: When triggered, the system **retrieves** snapshots from the buffer (from newest to oldest).

- **Smooth** approach: Interpolate states between snapshots over a short duration (e.g., one snapshot per RecordInterval) to produce a fluid motion backward in time.
- **Discrete** approach: Instantly apply the last snapshot, step by step.

4. **Core**: All logic (coroutines, event callbacks, ring buffer manipulation) is centralized in RewindableStructBase, so each specialized rewinder only defines **what** data to store and **how** to apply it.

Sequence of Operations

1. **Start Recording**

- A coroutine begins calling SetRecordSlot(...) at intervals (RecordInterval).

2. **Stop Recording**

- The coroutine ends, and no more snapshots are stored.

3. **Start Rewind**

- The system checks if snapshots exist; if yes, it begins removing them from the ring buffer to **restore** states in reverse chronological order.
- If SmoothRewind is true, it calls ApplyStateCoroutine(...). Otherwise, it calls ApplyState(...) directly.

4. **Stop Rewind**

- The rewind coroutine ends.
- Optionally, we can re-enter recording mode or remain idle.

- IRewindable: The interface guaranteeing StartRecord(), StopRecord(), StartRewind(), StopRewind() methods plus event callbacks.
- RewindableStructBase: Abstract class implementing IRewindable logic (managing ring buffer, coroutines, events).
- **Specialized Rewinders**: Implement how to capture data into TRewindData and how to apply it back.

3. Core Components

3.1 IRewindable

Key Role:

- Defines the fundamental contract for any rewindable object in the system.
- Offers event callbacks: OnRewindStartedCallback, OnRewindEndedCallback, etc.

3.2 RewindInfo

Key Role:

- Holds essential configuration:
- **RecordInterval**: Time between stored snapshots.
- **RecordCapacity**: Max snapshots in buffer.
- **SmoothRewind**: Whether to interpolate states.
- **RewindSpeed**: Speed multiplier for reversing.
- **RewindCurve**: The interpolation curve for smooth rewinds.

3.3 ICoroutineRunner & DefaultMonoBehaviourRunner

Key Role:

- Provide a mechanism to launch coroutines **without** tying logic to a specific MonoBehaviour script.
- DefaultMonoBehaviourRunner spawns a hidden GameObject ([DefaultMonoBehaviourRunner]) to host the coroutines if needed.

3.4 IRewindBuffer & StructRingBuffer

Key Role:

- IRewindBuffer<T> is an abstraction for buffer operations (add, remove, retrieve).
- StructRingBuffer<T> is the concrete ring buffer using an array and head/tail pointers.
- Ensures **fixed memory** usage by capping at RecordCapacity.

3.5 RewindableStructBase<T, TRewindData>

Key Role:

- **Abstract base** for all rewinding logic. Child classes define:

1. SetRecordSlot(ref TRewindData data): What data to store each frame.
2. ApplyState(ref TRewindData stateRefForApply): How to instantly apply that state.
3. ApplyStateCoroutine(TRewindData stateForApply): How to apply that state gradually (if SmoothRewind is enabled).
4. HasSmoothRewind(): Returns true if your child class can handle interpolation smoothly.

- Manages:
- **Record**: Creates a coroutine to store snapshots at RewindInfo.RecordInterval.
- **Rewind**: Creates a coroutine to restore snapshots in reverse order, optionally using interpolation.
- **Events**: OnRecordStarted/Ended, OnRewindStarted/Ended/Updated, OnBeforeBorn (if data goes before birth time).

3.6 Rewinder

Key Role:

- A “manager” for multiple IRewindable objects.
- Allows you to start or stop recording/rewinding all registered objects with a single call.
- Especially useful if you want a centralized point to coordinate multiple game entities.

4. Component Rewinders

4.1 TransformRewinder

- Records a Transform’s position, rotation, localScale.

- TransformData struct has exactly those fields.
- If HasSmoothRewind() is true, uses a coroutine to interpolate from the current to a previous transform state.

4.2 RigidbodyRewinder

- For 3D physics (Rigidbody).
- Captures positional data, velocity, mass, drag, isKinematic, etc.
- Often sets isKinematic to true during rewind to avoid physics interference.

4.3 Rigidbody2DRewinder

- Same concept, but for Rigidbody2D in 2D physics.
- Saves position, rotation, velocity, angularVelocity, gravityScale, etc.

4.4 ParticleRewinder

- Captures arrays of ParticleSystem.Particle.
- Optionally recurses into child ParticleSystems.
- During rewind, repositions/emits particles to reconstruct the recorded state.

4.5 AnimatorRewinder

- Specifically handles Animator states, including a HumanPose for humanoid rigs.
- Stores float/bool/int parameters plus the current AnimatorStateInfo.
- Can forcibly set the Animator's current state/time (Play(shortNameHash, ...)).

4.6 GameObjectRewinder

- Records basic GameObject-level properties: activeSelf, layer, tag, name, optionally parent transform.
- Useful for toggling active/inactive states or reverting to a previous transform parent.

5. Shared Data Structures

Each specialized rewinder defines a **struct** for the data it records:

- **TransformData**: (Vector3 Position, Quaternion Rotation, Vector3 Scale)

- **RigidbodyData**: (Position, Rotation, Velocity, AngularVelocity, Mass, ...)
- **ParticleData**: (ParticleSystem.Particle[] Particles)
- **AnimatorData**: (HumanPose Pose, Dictionary<string, object> Parameters, AnimatorStateInfo StateInfo)
- etc.

6. Usage

All these can reside in a **SharedTypes** folder for clarity.

1. **Create a Rewinder**

```
Instancevar transformRewinder = new TransformRewinder(myTransform, new RewindInfo(
recordInterval: 0.1f,
recordCapacity: 100,
smoothRewind: true,
rewindSpeed: 1f,
rewindCurve: null // defaults to linear
));
```

This spawns a [DefaultMonoBehaviourRunner] that will manage coroutines automatically.

2. **Start Recording**

```
transformRewinder.StartRecord();
// ...
transformRewinder.StopRecord();
transformRewinder.StartRewind();
// ...
transformRewinder.StopRewind();
```

3. **Start Rewind**

```
transformRewinder.StartRewind();
// ...
```

```
transformRewinder.StopRewind();
```

4. **Handling Events**

```
transformRewinder.OnRewindStartedCallback += () => Debug.Log("Rewind just started!");

transformRewinder.OnBeforeBornCallback += () => Debug.Log("No older data available.");
```

5. **Managing Multiple Objects**

```
var rewinderManager = new Rewinder();

rewinderManager.RegisterRewindable(transformRewinder);

// Possibly also register a RigidbodyRewinder, etc.

rewinderManager.StartRecord(); // all objects start recording

// ...

rewinderManager.StopRecord();

rewinderManager.StartRewind(); // all rewinding

// ...

rewinderManager.StopRewind();
```