RWND Documentation

RWND is a Unity plugin designed to allow the storage and restoration of game state over a period of time. The 3 main use cases for this are:

- Rewind - store game state and then rewind the entire game back to an earlier state. The game state flows backwards.
- Recall - store game state and rewind an individual object
- Replay - store game state and then play it back. The game state flows forwards from an earlier point.

API Documentation

[Rewind: Main Page (aeric.games)]

Rewind Components

Rewind components are central to the whole system. Each one is responsible for writing and reading data for a certain component type. For example RewindTransform is responsible for storing the position, rotation, and scale of a Transform component.

The existing rewind component type, and their corresponding source component types, are:

- RewindTransform (Transform)
- RewindAnimator (Animator)
- RewindCamera (Camera)
- RewindCustomMonoBehaviourAttributes (*)
- RewindGameObjectEnabled (GameObject)
- RewindMaterialColor (Material)
- RewindParticleSystem (ParticleSystem)
- RewindRendererMaterialChange (Renderer)
- RewindTransformHierarchy (Transform)

RewindCustomMonoBehaviourAttributes is a special case because it is designed to be inherited from and not used directly. There is more information about the component types in the API documentation.

Each rewind component declares a data schema that is used to:

- Define the size of the data needed for storage
- Convert the binary data to json

Creating a new Rewind component type

The system is designed to allow new rewind component types to be added relatively easily. There are 2 ways to go about this.

Inherit from RewindComponentBase

We will go over an example. Let's say we want the ability to alter the properties of SphereCollider components and rewind their state. We will add a new rewind component type called RewindSphereCollider.

1. Create the class, which will inherit from RewindComponentBase, and put in empty overrides for the various methods.

```
using UnityEngine;

namespace aeric.rewind_plugin {
    public class RewindSphereCollider : RewindComponentBase {
        private SphereCollider _collider;

        private void Awake() {
            _collider = GetComponent<SphereCollider>();
        }

        public override RewindDataSchema makeDataSchema() {
            return new RewindDataSchema();
        }

        public override uint HandlerTypeID => 11;

        public override void rewindStore(NativeByteArrayWriter writer) {
        }

        public override void rewindRestoreInterpolated(NativeByteArrayReader
frameReaderA, NativeByteArrayReader frameReaderB, float frameT) {
        }
    }
}
```

In the Awake method we cache any component references that we need. In this case that just means getting a reference to the SphereCollider component. HandlerTypeID should be set to a unique value that isn't being used by another rewind component type.

rewindStore and rewindRestoreInterpolated are the methods that are used to store and restore the state respectively. We will leave them blank for now and discuss the data schema.

The makeDataSchema method creates a data schema, which is an object that describes the state that we store for this rewind component. This is used to calculate the overall size of the data that we need to store, and it also enables some other features like being able to convert our stored binary data into json and other formats.

We first need to decide the data that we want to store. Looking at the SphereCollider component we decide that we want to store the center (Vector3) and radius (float). This means that makeDataSchema should look like this:

```
        public override RewindDataSchema makeDataSchema() => new
RewindDataSchema().addVector3().addFloat();
```

Next we will implement the rewindStore method. This is the method that is used by the system to store data for later use. All the data is stored in a native byte array, and is copied using low-level memory copying. If we want to store the center and radius of the SphereCollider then the method should look like this:

```
        public override void rewindStore(NativeByteArrayWriter writer) {
            writer.writeVector3(_collider.center);
            writer.writeFloat(_collider.radius);
        }
```

The other side of this is that we need to restore this data at some point. When we store data we store it in what we call a handler frame. We don't necessarily store a handler frame for every single frame of the game. Depending on our use case we might store only a few frames every second. But when we restore the data we want it to replay as smoothly as possible between the frames that we stored. This means that when data is restored it is always interpolated between 2 handler frames, the one immediately before our current time, and the one immediately after. The frameT value passed into rewindRestoreInterpolated tells us how far between those 2 frames we are. We use the frameT value to lerp each data point, and for our RewindSphereCollider component this will look like:

```
public override void rewindRestoreInterpolated(NativeByteArrayReader readerA,
NativeByteArrayReader readerB, float frameT) {
    _collider.center=Vector3.Lerp(readerA.readVector3(),readerB.readVector3(),frameT);
    _collider.radius = Mathf.Lerp(readerA.readFloat(), readerB.readFloat(), frameT);
}
```

And this is all we need to do! Our RewindSphereCollider can then be added onto any GameObject that has a SphereCollider component.

Inherit from RewindCustomMonoBehaviourAttributes

Inheriting from RewindComponentBase is a good fit for when you want to add rewind abilities to an existing component, especially a builtin one that you can't modify. It isn't always the best fit though. Sometimes you just want to add the ability to rewind certain fields of a particular component. An example of this is the RecallPlatform component used in the Recall demo. The movement of the platforms is controlled by interpolating between 2 points. To be able to rewind the movement of the platforms we need to rewind the value that controls this movement, which is the moveT field.

We can do this quite simply.

- Inherit from RewindCustomMonoBehaviourAttributes
- Also inherit from the IRewindDataHandler interface
- Add the [Rewind] attribute to any fields that we want to rewind. For example:
    - `[Rewind] private float moveT;`

Data Storage

All component data is stored in a native byte array. All writes are performed using low-level memory copies for the best performance.

Each rewind component is given a chunk of this array to write into. When we write a handlers data into the array we first write the id of the component, and then we write out the component data. This is usually done one value at a time, but there is support for writing byte arrays.

We currently support writing the following types:

- Float
- Vector3
- Quaternion
- Int
- Bool
- Color
- Unsigned int

Adding support for a new type involves changing several areas of code, but each change is quite simple.

1. Add methods to NativeByteArrayReader and NativeByteArrayWriter to read and write the type.
2. Add the type to RewindStorageData_Value.
3. Add the type to RewindDataPointType
4. Add the type to RewindDataSchema
5. Add support to RewindCustomMonoBehaviourAttributes

Serialization

At runtime all the game state is serialized to a native byte array, but we also support serializing to 3 types of file:

- Binary raw
- Binary stream
- Json

Demos

All the demos can be accessed via the menu in the Assets/rewind/Examples/scenes/menu.unity scene.

Replay

This demo shows a sports-game scenario, with 2 teams of robots attempting to capture points in the map. Triggering a replay shows the last several seconds of action, with a camera-in-camera that shows the most recent capture event. The game state is recorded for all objects and replayed from an earlier point. We use a RewindEventStream to capture game events for the point captures.

Recall

This demo shows an adventure-game scenario with a third person character navigating platform puzzles, and with the ability to rewind the platforms. The game state is recorded for all objects and the player has the ability to select and rewind a single object.

Rewind

This demo shows a sports-game third-person scenario where 2 teams of robots are competing to capture points in the map. At any time the player can rewind the last several seconds of gameplay.

Baking

This demo shows the ability to write out game state to 3 different file formats and then replay it later.