6

Debugging AI with Instant In-Game Scrubbing

David Young

6.1 Introduction

Catching AI in the act of making a wrong decision is vital in understanding what logic error occurred and what the root cause of the problem was. Unfortunately, the time between seeing a decision acted out and the actual act of making that decision can mean that all relevant information has already been discarded. Ideally if the entire game simulation could be rewound to the exact moment in time when the error occurred, it would make notoriously difficult problems to debug, trivial to understand why they occurred.

Game engines have typically made reproducing these types of problems easier using deterministic playback methods (Dickinson 2001), where the entire state of the game simulation can jump back in time and resimulate the same problem over and over (Llopis 2008). Unfortunately, retrofitting determinism into an engine which currently is not deterministic is nontrivial and requires meticulous upkeep to prevent inadvertently breaking determinism.

Luckily, debugging AI decision-making and animation selection after the fact does not mandate resimulation or even reproduction of the issue. Using an in-game recorder system provides all the flexibility necessary to record all relevant data as well as allowing the visual representation of the game simulation to be stepped back and forth in time to understand when, where, and why the error occurred. This chapter will lay out the architecture of the in-game recorder system used internally in the creation of AAA games developed by Activision Treyarch.

6.2 In-Game Recorder

At a high level, an in-game recorder is a relatively straightforward concept; after each simulation frame finishes, iterate through every game object and record the minimal amount of data necessary to reproduce the exact visual state of the object. To replay recorded data, pause the game simulation, and restore all recorded data of each recorded game object from a single simulation frame. In order for a recorder system to be practically useful though during development, other real-world considerations must be met. The recording of object data must have minimal to no impact on the speed of the game simulation, the memory requirements of the system must be fixed, and the recording of object data must have no impact on the outcome of simulating objects. With these requirements in mind, the remainder of the chapter will detail each of the main components used to create an in-game recording system, as well as the practical uses of debugging AI using recorded debug information in tandem with navigating through playback, also known as scrubbing through playback.

6.2.1 Recorder Architecture

The most fundamental building blocks of recorder data are individual packets of data stored about recorded game objects. After a single frame of the game simulation finishes, a single packet of recorder data is constructed per object storing information such as position, orientation, joint matrices, and visibility. The entirety of all packets recorded per simulation frame will fluctuate and are collectively stored within a frame of recording data. Each recorded frame is timestamped with the absolute time of the simulation as well as the relative time between the last recorded frame to facilitate quick retrieval and accurate playback.

All recorded frames of simulation are stored and managed by a single recorder manager, which provides the interface between interacting with the recorder system as well as updating each subsystem to record object data. The manager internally stores a fixed-sized memory buffer which allocates and deallocates the memory consumed per packet and per frame of recorder data. When no additional free memory is available for allocation within the memory buffer, the oldest recorded frames of data are freed in order to allocate enough memory necessary to store a new frame's worth of packet data.

The responsibility of serializing recorder data is left to individual record handler object implementations that only serialize record data based on the type of object they are classified for. In practice, only a few varieties of record handlers as necessary to cover recording animated models, first person view models, player characters, and player cameras. To handle the playback of recorded data, playback handler objects implement deserialization functionality based on the type of object they are classified for. Typically, there is a one-to-one mapping of record handlers and playback handlers registered for use by the recorder.

The last component of the system deals with the user's interaction during record playback. A specialized controller implementation overrides the systems default controller implementation when recorder playback is enabled. The controller's button scheme is configured to provide access for free cam movement as well as rewinding, fast forwarding, and stepping through individual frames of recorded data. Additionally, the controller scheme allows for selecting particular objects of interest to limit rendering of additional debug information from superfluous objects. Figure 6.1 shows the

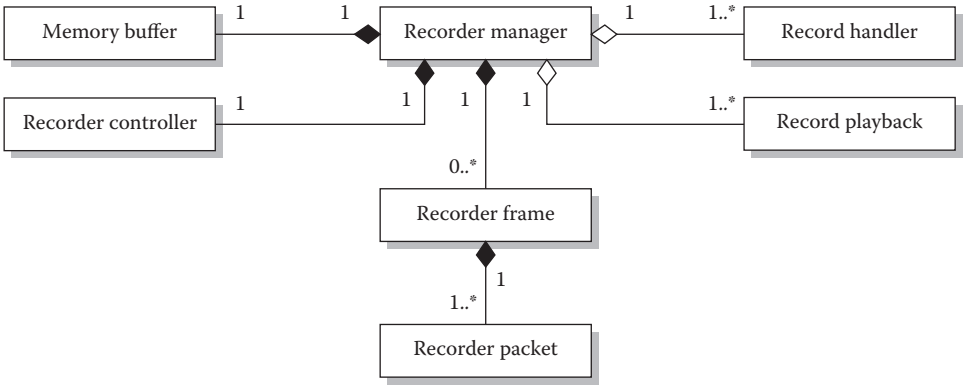

Figure 6.1

High-level class diagram of the recorder architecture.

high-level relationships between each subsystem of the recorder's architecture as well as the ownership of internal data used by the system.

6.3 Recording Data Primitives

Since a recorder is only a visualization of the past, when constructing primitive packet data, only the minimal visual information needed to reconstruct the state of the game object is necessary to be serialized. Typically, at a minimum, this piece of data includes the game object id, the type of the object, the position of the object, and if the object is targetable during recorder playback. Figure 6.2 shows examples of different types of recorder packet information based on what type of primitive is being recorded.

Looking at a particular packet type, the `RecorderModelPacket`, let us dissect what exact information is being stored per game object. The first data member stored about the game model is a unique identifier representing the game object available throughout

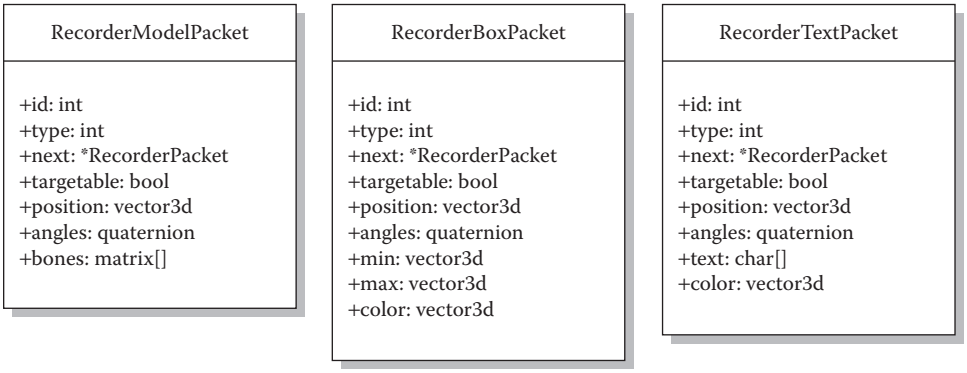

Figure 6.2

Example data structures of different recorder packet types.

the engine. Although no information about the particular model is stored in the packet, retrieval of this information is available with the use of the unique id. The next field stored defines what type of object the id corresponds to and is similarly used through the engine to classify specific game objects.

A pointer to the next packet is stored within each recorder packet to form a singly linked list of all packet data within a frame. A singly linked list of packets is used as the internal data structure for recorder frames since individual packets can have varying sizes depending on the game object they are serializing. For instance, recording model data is dependent on the number of bones a particular model has, which varies model to model even though the game object type is the same.

The next field represents if the model should be selectable to toggle additional recorded debug information during recorder playback. The position field is self-explanatory and represents the current position of the object within the world. The angles field represents the current orientation of the model. The last field represents the current orientation and position of all skeletal bones the model has. If the model has no skeletal bones, this field is left null; otherwise an array of bone matrices is stored. Although it may seem counterintuitive to store a verbose representation for a game object's entire skeleton, storing animated bone matrices is preferable to storing animation data directly in order to account for physically driven or inverse kinematic bone constraints.

In addition to storing game object data, the recorder provides storing additional debug only primitives, which provide much of the debug capabilities of using a recorder system. Exposing a multitude of debug primitives such as lines, spheres, boxes, polygons, and text primitives associated with specific game objects allow for easily visualizing the underlying decision-making structures of an AI system. During normal game simulation, rendering the same debug primitives submitted to the recorder allows for identical debug capabilities during runtime as well as when scrubbing through recorder playback.

While recording every game object in this manner can be extremely useful during development, selectively recording only game objects of interest can greatly extend the amount of recording time allotted by the fixed memory budget. The use of recorder channels' facilities allowing users to select what type of objects should be recorded at any given time within the system. Each game object type can be classified to belong in one or more channel classifications such as AI, animation, script, and fx. Debug primitives can also specify which channel they should belong to so they appear appropriately when debugging specific systems such as animation or scripting.

6.3.1 Record Handlers

Since each game object type typically has dramatically different requirements for serializing visual representation, the recorder exposes a simple interface to implement different record handlers. A record handler primarily serves two different functions within the recorder: calculating the amount of memory required for any specific game object instance as well serializing the game object type within the specified allocated memory. Each record handler instance is in turn registered with the recorder specifying the type of game object the handler is able to serialize.

```
int GetRecordSize(entity* gameObject);
void Record(replay_packet* memory, entity* gameObject);
```

When a game object is being processed by the recorder for serialization, the corresponding record handler is first queued about the amount of memory required to construct a replay packet of information about the game object. If enough free memory exists within the memory buffer, the recorder will immediately call `Record` passing in the requested memory; otherwise the recorder must first deallocate the oldest recorded data within the memory buffer until the requested size can be allotted.

Decoupling serialization from the recorder's main responsibilities allows for extending the recorders support for new game object types within the game engine without having to modify the recorder system itself. In particular, new game object types are easily implemented by creating a new record handler and assigning an instance of the record handler during recorder registration.

One area to note though is the opaque nature of the record handler memory system. Replay packet memory passed into the `Record` function is merely uninitialized memory passed from the recorder's memory buffer. The recorder itself only knows how to interpret the initial recorder packet fields: id, type, next, targetable, position, and angles shared between all record packets but does not know how to interpret the remaining additional memory within varying packet types. The record handler must work in tandem with the playback handler in order to properly interpret the additional memory.

6.4 Recording Simulation Frames

All recorder packets from a single simulation frame are stored within a single recorder frame structure. Figure 6.3 shows the data structure of a typical recorder frame instance. Looking at the fields of a recorder frame, the first field, id, represents the unique identifier of the recorder frame. The next two fields store pointers to the previous recorded frame as well as the next recorder frame if one exists. Since frame time may vary and correctly associating a recorded frame to a specific timestamp is critical within the recorder system both the absolute time of the simulation as well as the delta time since the last recorded frame are stored in order to play back recorded frames with the same consistence as the original user experienced. The last field, packets, points to the head of the singly linked list of recorder packet data that are associated with the frame. As traversal of packet data is always linear, a linked list was sufficient for fast access times.

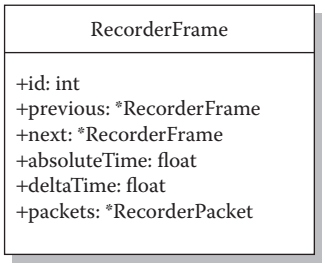

Figure 6.3

Recorder frame data structure.

Managing each simulation frame of recorder data primarily focuses around fast access of adjacent frame data when traversing frames forward or backward in time. Each newly recorded frame of data stores a head pointer to the first packet of recorder data as well as a pointer to the previous and next address of additional simulation frame recordings. Using a doubly linked list data structure allows for minimal upkeep when frames are added or destroyed since new frames are always added to the beginning of the list, and deleting frames are pruned from the tail end of the list.

To enable proper playback of recorded frames, storing both the absolute time of the game world as well as the delta time since the last frame was recorded provides the ability for determining which recorded frame of data represents any specific moment of time within the game world as well as accounting for variances in time step. Even though most game simulations run at fixed time steps, the actual amount of time for a simulation frame may exceed the allotted time budget. Storing a relative delta time within each frame allows for replaying data at the same rate as the original capture rate.

Even though both an absolute and relative timestamp is stored, other system's interactions with the recorder are done strictly based on game time for simplicity. Calculating the correct game time utilizes the stored absolute times within each frame of data, while scrubbing backward and forward in time utilizes only the delta time stored within a frame.

6.4.1 Recorder Manager

Managing the recorder system revolves around two distinct responsibilities, recording data after a simulation frame through a standard polling update loop and replaying recorded data when scrubbing through recorded frames. Figure 6.4 shows the high-level class diagram of the recorder manager's member data and external interface.

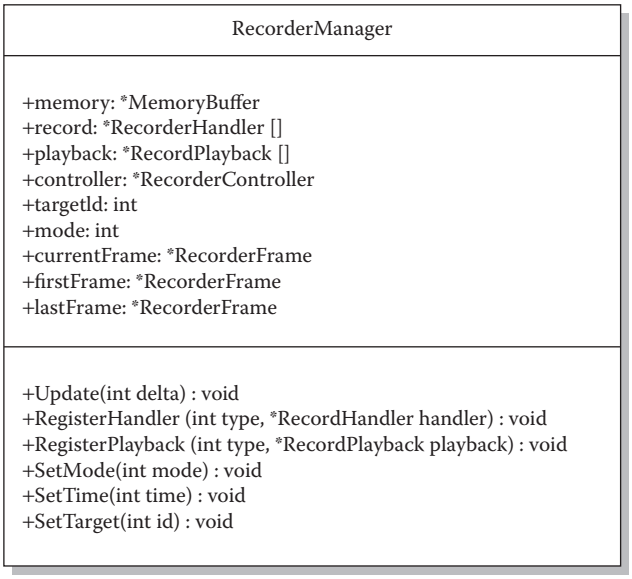

Figure 6.4

Recorder manager class diagram.

Starting with one of the most critical internal tasks within the recorder manager is the correct allocation and deallocation of recorder packets and frames. Internally, the manager uses the memory buffer to server as a circular buffer where the oldest recorded frames are discarded to allow new frames to be recorded. Initialization of the manager begins with allocating a contiguous memory block to become the memory which the buffer internally manages. Registration of both record handlers and playback handlers soon follows before the manager begins recording game object data.

The game engine's overall management and update ticking of the recorder system are handled by a single instance of a recorder manager. The recorder manager's update function is called after the end of the game simulation loop to ensure that all processing of game objects has been completed before record handlers process each game object. As the recorder manager processes each game object, the game object is first classified based on object type, and the corresponding record handler is used for determining the proper amount of memory to request from the buffer. Once memory has been allocated, the recorder manager passes both the allocated memory and game object directly to the record hander for serialization. After a packet of information is serialized, the manager will correctly fix up each packet pointer as well as assigning the frame's starting pointer to the first packet of recorded information.

Once all packet data within a recorder frame are serialized, the manager will update the corresponding next and previous frame pointers of the current first frame of recorder data to attach the newly recorded frame of data to the doubly linked list. The new frame of data now becomes the first frame of data for the manager to use when playback is requested.

Additional information is stored within the manager to allow quick access to both the most recent recorded frame of data as well as the oldest recorded frame of data for scrubbing as well as updating the doubly linked list when frames are deallocated and new frames are allocated. During recorder playback, the current frame of data being replayed is stored separately to allow for quick resuming of in-game scrubbing as well as selectively stepping forward and backward.

6.4.2 Memory Management

Since the recorder continuously runs in the background working within a fixed memory budget, whenever the memory buffer is unable to allocate a requested amount of memory, the recorder manager will continuously deallocate the oldest recorded frame until enough free memory is available to accommodate the new allocation request. Continuously deallocating the oldest recorded frames and replacing those frames with new frame data achieve minimal memory fragmentation within the memory buffer, causing the buffer to act as a circular buffer, even though only contiguous memory is used.

Figure 6.5 shows the state of memory as frames are continuously added to the buffer until the amount of free memory remaining would not be sufficient for an additional frame of recorder data. In this case, Figure 6.6 shows how the memory buffer handles the deallocation of the first frame of stored data, Frame 1, being immediately replaced with Frame 3's data. Acting as a circular buffer, the only portions of unable memory with this setup will be between the last recorded frame and the first recorded frame as well as some portion of unusable memory at the tail end of the memory buffer's allotted memory.

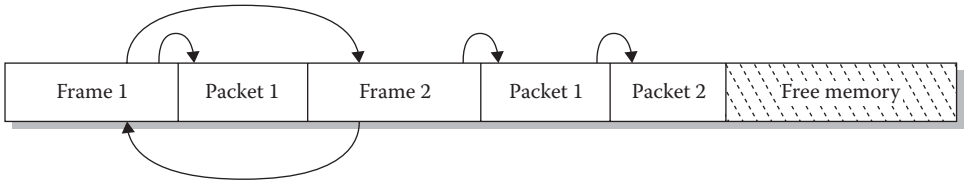

Figure 6.5

Contiguous memory layout of recorder frames and frame packets.

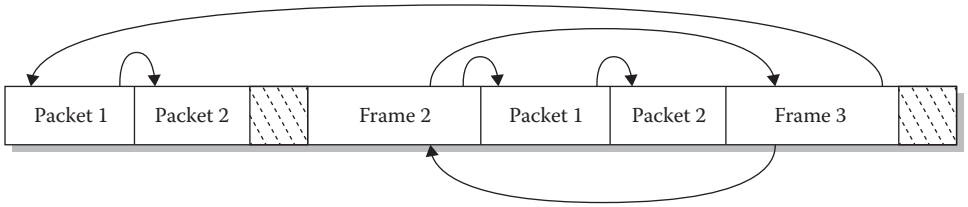

Figure 6.6

Memory buffer with minimal unusable memory and fragmentation.

6.5 Frame Playback

In order to play back recorded frames, a few considerations need to be made to prevent affecting the game simulation itself. Since restoring recorded data such as position and orientation will act directly on a game object's current state, the recorder must wait till the end of recording the latest frame's packet data before pausing the game simulation and overwriting game object data. Inversely before the recorder exits playback mode, the last recorded frame data must be reapplied to each game object to leave every game object in the same condition before playback mode was enabled.

When playback mode is requested to the recorder manager, the simulation pauses without any other work being necessary from the recorder unless an external source has requested the recorder to jump back in time to a specific timestamp. In this case when traversing frame data based on a timestamp, the recorder will start with the latest recorded frame data and work backward within the linked list of data to find the frame either matching the exact requested timestamp, the first timestamp immediately before the requested time, or the oldest remaining timestamp within the linked list.

Scrubbing forward and backward in time works on a completely different principle than the game world's time. Scrubbing can either occur stepping a single frame of data forward or backward or can replay data at the exact same rate as the original recording. When scrubbing in real time, the recorder's update loop will maintain a delta time since scrubbing was requested and continuously move backward or forward replaying frame data based solely on the accrued delta time of each frame's data. Managing time delta based on the original recording rate allows for frames of data to be replayed regardless of the game's delta time, which may differ from the actual delta time between each frame of recorder data.

6.5.1 Controlling Recorder Playback

Once in playback mode, overwriting the standard controller scheme with a custom controller scheme provides the necessary flexibility to manipulate playback. In particular, at least two speeds of operation for fast forwarding and rewinding are crucial for interacting with playback. Being able to play back data at a normal game simulation rate allows for fine tuning animation work while stepping the recorder one frame at a time is typically necessary for debugging decision-making logic. Since behavior selection is typically determined on a specific frame of simulation, all relevant recorded debug primitives are typically only drawn during a single frame of recording. In addition, the controller scheme is used for selectively targeting which game object's debug primitives are rendered during playback, which helps narrow debugging of any specific game object.

6.6 Debugging AI

AI in previous Treyarch titles used a recorder system extensively for animation, decision-making, and steering debugging. To debug animation playback and selection, an AI's entire animation tree was recorded with each of the AI's animation contribution, rate, normalized time, and selection criteria available. Scrubbing back and forth in time was pivotal for debugging animation pops, incorrect animation blends, and incorrect animation selection.

Debugging decision-making with the recorder allowed for recording an AI's entire behavior tree in game, which displayed which branches within the tree were currently executing as well as where within other possible decision branches, execution terminated prematurely. Whenever an AI exhibited an incorrect behavior or lacked selecting the correct behavior, it was trivial to immediately drop into recorder playback; scrub backward in time, and immediately understand which branch of the behavior tree did not execute as expected.

Debugging steering-related bugs with the recorder typically involved understanding and addressing undesired velocity oscillations or incorrect animation selections, which caused AI to run into each other or other obstacles. Rendering both the velocity, steering contributions, as well as projected animation translation of an AI frame by frame visually, quickly narrowed down incorrect AI settings, incorrectly authored animations, and malicious scripting overriding normal AI movement behavior.

6.6.1 Synchronizing External AI Tools

In addition to in-game debugging with the recorder, external AI tools tapped directly into the recorder's current timestamp. Using a custom message bus system, the recorder would broadcast what current timestamp was being scrubbed, which allowed other tools to synchronize playback of externally stored information with the recorder's visual playback. Utilizing the same message bus, the recorder could also be controlled externally through other tools to enter playback mode and jump to specific timestamps. In addition, a new type of assert was added to the engine called a recorder assert, which would pause the game simulation and set the camera immediately to a particular game object notifying all other external tools to synchronize with the targeted game object through the recorder.

6.7 Conclusion

Debugging the root cause of any AI bug can be a challenge; let alone during development where the state of the world is constantly fluctuating and new additions, modifications, and removals are occurring simultaneously. Although the use of a recorder is not a panacea and cannot stop bugs from being added, the ability to debug directly in production levels without the need to reproduce the issue in a test map once seen by the user is an incredibly powerful tool that saves countless man hours. No logging of debug information or text output can even begin to compare with the immediateness of being able to visualize every frame of game simulation both backward and forward in time. Past and present development of AI at Treyarch now uses a recorder as a central tool around which other tools and processes are created, and the benefits of such a system are still being discovered.

References

Dickinson, P. 2001. Instant replay: Building a game engine with reproducible behavior. *Gamasutra*. http://www.gamasutra.com/view/feature/131466/instant_replay_building_a_game_.php

Llopis, N. 2008. Back to the future. *Game Developer Magazine*. http://gamesfromwithin.com/back-to-the-future-part-1,http://gamesfromwithin.com/back-to-the-future-part-2 (accessed June 11, 2016).