DemoNetDriver and Streamers

Overview of the Replay system for recording and playback of gameplay

The `DemoNetDriver` uses **Streamers** to extract and record the information needed to create replays. There are several different Streamers included with the Engine that can be attached to the `DemoNetDriver` depending on how the replay data is intended to be viewed. The **Local File Streamer**, which is the default, records events from the host machine directly to disk, making it best suited for single-player games and games that keep replays locally, on the host player's own machine. A variant of the Local File Streamer, the **Save Game Streamer**, records replays into save game slots, which is especially useful on consoles, and supports supplemental features to play and manage these replays. The **Memory Streamer** runs on a client machine and stores data in memory, making it ideal for "instant replay" features in sports titles or "kill cams" in shooters. Finally, the **HTTP Streamer** is used to send replay data to a second machine over LAN or the Internet, which makes it a great choice for dedicated-server games and for games that may need to be streamed live to a large number of viewers while still remaining responsive to players.

DemoNetDriver Features

- DemoNetDrivers use the Local File Streamer by default, but can override this by receiving the URL option "ReplayStreamerOverride" set to the name of a different Streamer Factory's module, such as "InMemoryNetworkReplayStreaming" or "HttpNetworkReplayStreaming". The default value, "LocalFileNetworkReplayStreaming", can be changed by setting the "DefaultFactoryName" variable in the "NetworkReplayStreaming" section of your project's `DefaultEngine.Ini` file. This can also be accomplished by calling `InitBase` and providing the appropriate URL argument as a parameter.
- DemoNetDrivers can amortize the time cost of recording replay data by setting the "demo.CheckpointSaveMaxMSPerFrame" CVar to a positive value. Actors that are not recorded into the replay before the per-frame time limit expires will be queued to record on the next frame. The advantage to this feature is that it caps the amount of time that checkpoint recording can take and helps to keep your game free of hitches. The drawback is that slight visual errors can appear during playback due to checkpoints containing data on Actors taken from different frames. This feature will only be activated in the case that your game is taking longer than your specified time limit to record checkpoints, which means that it will primarily apply to lower-end machines, or to more performance-intensive games.
- If `bPrioritizeActors` is set to true, Actors being saved into a replay will be pre-sorted by priority for recording order, based on the virtual function `GetReplayPriority`. This is useful when combined with amortized recording via `MaxDesiredRecordTimeMS`.
- Checkpoint recording frequency can be adjusted by changing the CVar `demo.CVarCheckpointUploadDelayInSeconds`. The default is 30 seconds. Increasing the length of time between checkpoints will make scrubbing backward or skipping around in replays slower, but will decrease the size of replays.
- The variable `bPauseRecording` can be set to true while recording a demo to suspend recording temporarily. Setting it back to false will resume the recording.
- The **Game Mode** will use a different **Player Controller** class (designated as `ReplaySpectatorPlayerControllerClass`) when viewing replays.
- Using `SetViewerOverride`, a `DemoNetDriver` can change how Actors' network relevancy, culling, and prioritization are determined by creating an alternate **Player Controller** that will be used for recording purposes. This is especially useful for games with large maps, where the player isn't always aware of things happening far away during live play (both for efficiency and cheat-prevention reasons), but will expect to see everything when viewing a replay.

- DemoNetDrivers can operate in parallel with Slate. To accomplish this, both "tick.DoAsyncEndOfFrameTasks" and "demo.ClientRecordAsyncEndOfFrame" CVars must be non-zero.

> Replay-generated Actors will make function calls just like live-gameplay Actors do. This causes them to behave just like live actors with minimal replay data, but it also means that function calls that affect shared objects such as the `GameInstance`, `GameState`, or `GameMode`, will still be usable by replay Actors, and can affect the state of the game in unintended ways. This is especially true in the case of the Memory Streamer, which can view a replay while live gameplay is still running. To guard against Actors affecting things that they shouldn't, it is recommended that these operations first be checked to see if the Actor is part of a live or replay level, and behave accordingly. This is most likely a game-specific issue and must be handled by each project on a case-by-case basis; for example, a given game may wish to update the player's health bar or full-screen damage overlay during a replay, but not alter the player's score.

Replay Data Format

In terms of data, a replay contains three types of game-state information, as well as some additional text data. At the start is baseline data describing the starting state of the game world. Checkpoints that act as snapshots of the net changes to the world (from the baseline) appear at regular, user-defined intervals. The space between checkpoints is then filled with incremental changes to individual objects in the world. Any moment in the game can be reconstructed by the engine quickly and accurately by initializing the world to the starting state, making the changes described in the checkpoints before the chosen time, and then applying each incremental change after the most recent checkpoint leading up to the desired point in time. The text data contained in a replay consists of a display name, which can be used when making a player-facing list, and user-defined text tags (HTTP Streamer only), which can be used when searching or filtering through lists of games.

Local File Streamer

The Local File Streamer records replay data asynchronously to a single file on local storage (such as a hard drive). This is the default streamer as of its introduction in Engine version

4.20, replacing the Null Streamer. The Local File Streamer's single-file output makes distribution and management of saved replays easier, and its asynchronous recording improves performance on systems with lower hard drive speeds, such as consoles. Replay data files are saved to the "Saved/Demos/" folder in your project, and have the extension ".replay".

> The Local File Streamer is not compatible with replays recorded by the Null Streamer. If your project uses the Local File Streamer, but you want to maintain backward compatibility with replays recorded by the Null Streamer, you will need to include the Null Streamer. This does not preclude the project from using the Local File Streamer moving forward.

SaveGame Replay Streamer

The SaveGame Replay Streamer is a specialized version of the Local File Streamer that adds the ability to transfer a replay into a **savegame** slot. This is particularly useful on consoles, where client-side replays can be saved and then loaded through the platform's normal interface for loading saved games. Although this Streamer retains the ability to store, save, and load replays without the savegame system, its main purpose lies in its supplemental API, which can identify replays that haven't been copied to a savegame slot, perform the copy, and play or delete replays, both from local files and from savegame slots.

The Null Streamer

The Null Streamer writes replay data directly to disk, such as a local hard drive. This is good for making local recordings, especially of single-player games. These recordings can be very helpful for a variety of uses, such as producing gameplay trailers or in-game cutscenes, or enabling your users to view and share speedrun or tutorial videos within your game. Replay data files are saved to the "Saved/Demos/(Replay Name)" folder in your project.

> The Null Streamer was the default way to record a replay before Engine version 4.20. It has since been deprecated, but can still support backward compatibility by playing old replays.

The Memory Streamer

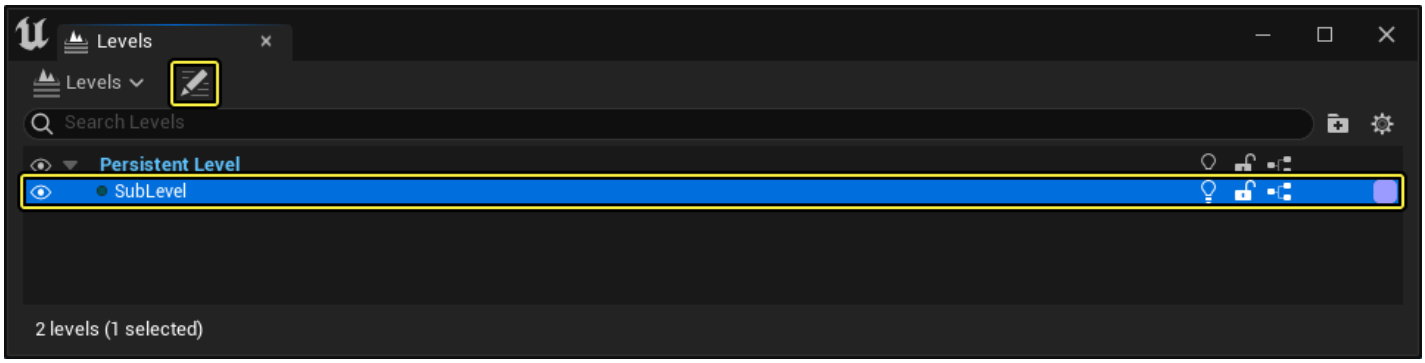
The Memory Streamer keeps a user-configurable running length of replay data in memory on the local machine. This type of stream is best suited to instant replays of recent, dramatic moments, such as scores in a sports game, deaths in a shooter, or the final moments of a boss battle in an action game.

Memory Streamer Usage Details

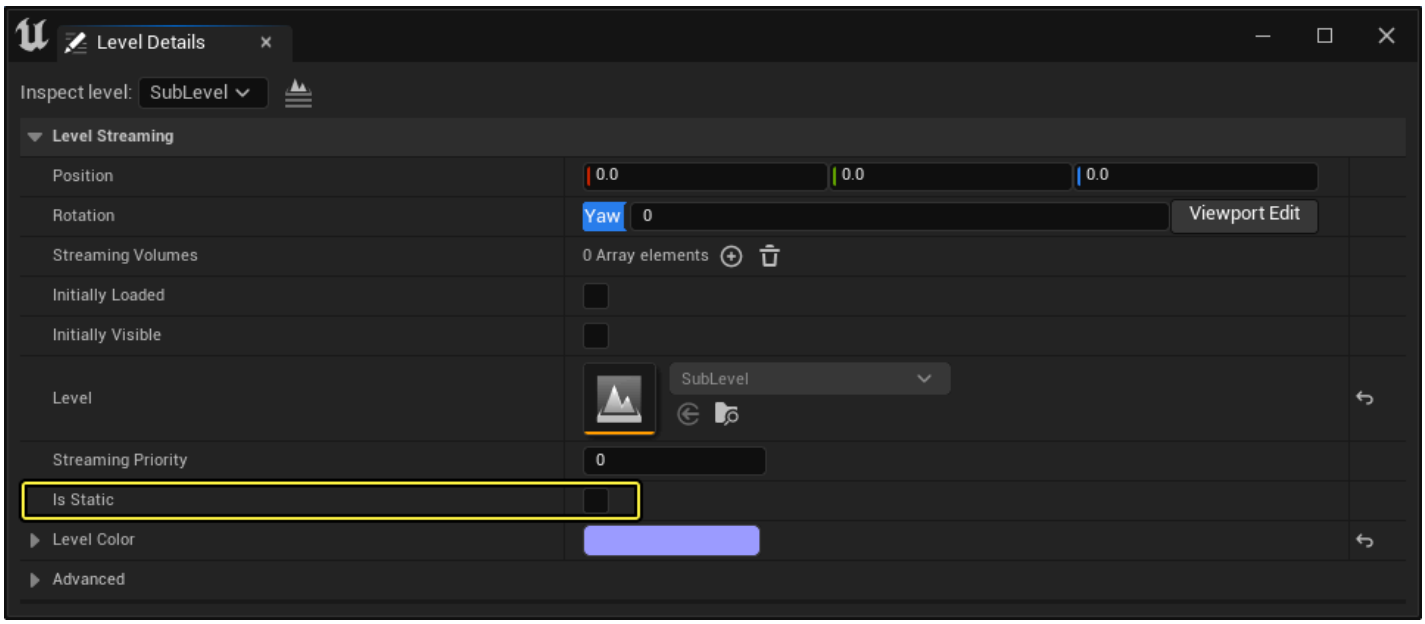
The Memory Streamer is special in that it is meant to record, play back, and resume gameplay during a single session. The live game is able to continue, invisibly and silently, while the player is watching a replay, so that the game can be resumed seamlessly the moment the replay ends. At load time, the engine collects levels into three groups: Static Levels, Dynamic Source Levels, and Dynamic Duplicated Levels. These groups determine how the level will interact with live gameplay and the Replay System, as follows:

Level collection	Levels added to this collection	Behavior
Static Levels	Levels that are not the persistent level and are marked with the `IsStatic` boolean.	Should be unaffected by gameplay, and will be shown during both live play and replays.
Dynamic Source Levels	The persistent level and any sublevels with the `IsStatic` boolean variable set to false.	Affected by live gameplay. Hidden during replays, but with gameplay still running as normal.
Dynamic Duplicate Levels	Copies made from Dynamic Source Levels at load time. Do not exist on dedicated servers or in editor mode.	Hidden during live gameplay. Replays take place in these levels, then they are emptied out.

The "Is Static" setting can be found by activating the Levels view from the Window menu, and then clicking the Level Details button with your sublevel highlighted.



From there, the "Is Static" option will be available in the Level Details window. This window will be empty for the persistent level, so make sure to select a sublevel.



> These level collections are a part of the level-streaming process and are not specifically related to the Replay system.

With this system in mind, we can create one `DemoNetDriver` for the Dynamic Source Levels, and another for the Dynamic Duplicate Levels. This enables us to record live gameplay in the Dynamic Source Levels and then play that data back in the Dynamic Duplicate Levels. By hiding the Dynamic Source Levels and showing the Dynamic Duplicated Levels during replay, the game can continue playing and receiving network updates unaffected by the replay. The third group, Static Levels, can be active and visible at any time; they should contain things like static world geometry or ambient background sounds, particles, and animations that are not affected by live gameplay, and therefore do not need to be involved in the replay process. When the replay is finished, the contents of the Dynamic Duplicate Levels will be destroyed and will undergo garbage collection, and the Dynamic Source Levels will become visible/audible again. Since the Dynamic Source Levels were never destroyed or suspended,

only hidden, the game will have naturally progressed while the replay was being viewed, and can be shown immediately and without hitches. Additionally, this system enables increased efficiency by giving the developers the ability to mark levels as static in order to exclude them from replay recording and playback, saving both memory and time.

The HTTP Streamer

The HTTP Streamer sends replay data to another server, which could be on a LAN or elsewhere on the Internet. This is useful for live-streaming matches, or for keeping recordings of matches that can be viewed at any time. This streamer is especially useful for dedicated-server games, where only the server knows everything that's happening everywhere in the game at all times, and where offloading the work of processing replay data will increase the number of simultaneous games that can be hosted on a single server. It can also help to serve as a moderation or cheat detection tool, since the data can be captured from computers completely controlled by the party running the game.

HTTP Streamer Usage Details

The HTTP Streamer will communicate with a custom-written replay server through the HTTP Streamer REST API, using GET and POST methods to send data as binary or JSON-formatted strings. In order to set up your own replay server, you must first establish your URL. Your project's DefaultEngine.Ini file contains the replay server's URL under the `[HttpNetworkReplayStreaming]` section, in the variable `ServerURL`. Your `ServerURL` should be in the format "http://replay.yourgame.com/". The final "/" is important, as the HTTP Streamer will not assume that it should modify the format of the URL.