Libretro - Implementing the core

Hans-Kristian Arntzen, Daniel De Matteis

December 3, 2012

Contents

Abstract

This document explains how to successfully implement a library based on the libretro API. Several reasons for why more emulators and game engines should run in libretro are outlined.

1   Background

RetroArch is a project that aims to be a center for "retro" gaming experiences. For your quick fix of 2D gaming goodness, RetroArch aims to provide a simple, featureful uniform interface for many different gaming systems.

Libretro is an API that abstracts the inner functionality of a gaming system. RetroArch can load a library that implements this API, give it a ROM, and play. A key design is that libretro implementations are loaded as libraries. This ensures great modularity, and flexibility for a developer of a core.

Due to its simple and open API, other frontends can just as easily utilize libretro implementations.

While "retro" would often imply an emulator of a classic system, libretro can also abstract a game engine. Classics such as Cave Story (NXEngine) and DOOM (PrBoom) have already been ported over.

2   Portability

Most emulator authors write both the backend and frontend to their project. The question of portability inevitably rises when the frontend is developed. Should one target a single platform with high level of integration or take a multi-platform approach with libraries like Qt? Backends for essentials like video, audio and input take a lot of time and effort to get right, especially on multiple platforms.

By implementing for libretro, one can target standard C/C++ (or any language that can export a C API), and achieve instant portability across tons of platforms.

In 2012, we would argue that portability has never been more relevant. There are three major operating systems on the desktop, two major smart phone platforms, and three gaming consoles. If your system is implemented with software rendering, your implementation can run on all systems supported by the frontend, without writing any platform specific code. In the case of RetroArch these are currently:

- Windows

- Linux

- OSX

- GameCube

- PlayStation 3

- XBox 1

- XBox 360

- Wii

- Android

3   Features

RetroArch has been in development for about two years. During this time, some features have shown to be quinessential to retro gaming systems. Implementing the libretro API correctly, these features can be utilized without any additional work.

3.1   Frame-by-frame rewind

A libretro implementation that implements serialization and unserialization of internal state is able to transparently support rewind mechanics. While many emulators support coarse grained rewind, RetroArch supports rewind at the frame level, i.e., frames can be rewound one frame at a time, similar to the indie-title Braid.

3.2   FFmpeg lossless recording

RetroArch can utilize the libavcodec library to encode video and audio output from a libretro implementation. The data is encoded losslessly, with FLAC as audio codec, and cutting-edge H.264/RGB (libx264) encoding, with a fallback to FFV1 for older playback systems that don't support the modern H.264/RGB variant. The recorder is multithreaded, and easily performs real-time.

3.3   Advanced GPU shader support

Classic 2D games have the advantage that their video output is very flexible, that is, it can be post- processed easily. Before the advent of programmable GPUs, video filters had to be performed on the main CPU, which cut directly into performance, and severely limited the types of filters possible to acheive in real-time. RetroArch aims to move this processing to the GPU by using shaders. A vast amount of shader effects are written already, and the shader format used is documented and implemented independently by other frontends as well.

RetroArch's shader support is more advanced than most emulators. It supports an arbitrary amount of shader passes, look-up textures (borders), scriptable shaders that can react dynamically to input or game content.

RetroArch still supports use of traditional CPU filters, however, it should be considered a fallback if GPU support is broken.

3.4 Peer-to-peer netplay

RetroArch supports two-player action over the network. It employs a rollback technique that aims to hide latency as much as possible, similar to the method employed by GGPO.

In addition to head-to-head multiplayer, a spectator mode is implemented. This mode allows a host to live stream playback to several watchers in real-time. The bandwidth required for this mode is near- zero as only raw input data is transferred over the network.

3.5 Audio DSP plugins

RetroArch supports plugins that allows post-processing of audio data, similar to post-processing of video data. It supports use of on-the-fly configurable plugins with aid of a GUI.

4 Implementing the API

The libretro API consists of several functions outlined in libretro.h, found in the RetroArch source package. A libretro implementation should be compiled into a dynamically loadable executable (.dll/.so/.dylib) or a static library (.a/.lib) that exports all the functions outlined in libretro.h. These will be called by the frontend. Implementations are designed to be single-instance, so global state is allowed. Should the frontend call these functions in wrong order, undefined behavior occurs.

The API header is compatible with C99 and C++. From C99, the bool type and <stdint.h> are used.

The program flow of a frontend using the libretro API can be expressed as follows:

4.1 Startup

4.1.1 retro_api_version()

This function should return RETRO_API_VERSION, defined in libretro.h. It is used by the frontend to determine if ABI/API are mismatched. The version will be bumped should there be any non- compatible changes to the API. Changes to retro_* structures, as well as changes in publically visible functions and/or their arguments will warrant a bump in API version.

4.1.2 retro_init()

This function is called once, and gives the implementation a chance to initialize data structures. This is sometimes implemented as a no-op.

4.1.3   retro_set_*()

Libretro is callback based. The frontend will set all callbacks at this stage, and the implementation must store these function pointers somewhere. The frontend can, at a later stage, call these.

4.1.4   Environment callback

While libretro has callbacks for video, audio and input, there's a callback type dubbed the environment callback. This callback (retro_environment_t) is a generic way for the libretro implementation to access features of the API that are considered too obscure to deserve its own symbols. It can be extended without breaking ABI. The callback has a return type of bool which tells if the frontend recognized the request given to it.

Most implementations of libretro will not use this callback at all.

4.1.5   retro_set_controller_port_device()

By default, joypads will be assumed to be inserted into the implementation. If the engine is sensitive to which type of input device is plugged in, the frontend may call this function to set the device to be used for a certain player. The implementation should try to auto-detect this if possible.

4.1.6   retro_get_system_info()

The frontend will typically request statically known information about the core such as the name of the implementation, version number, etc. The information returned should be stored statically. If dynamic allocation must take place, the implementation must make sure to free this storage in retro_deinit() later.

4.1.7   retro_load_game()

This function will load a ROM that the implementation will use to play the game. If the implementation is an emulator, this would be a game ROM image, if it is a game engine, this could be packaged upassets for the game, etc. The function takes a structure that points to the path where the ROM was loaded from, as well as a memory chunk of the already loaded file.

There are two modes of loading files with libretro. If the game engine requires to know the path of where the ROM image was loaded from, the need_fullpath field in retro_system_info must be set to true. If the path is required, the frontend will not load the file into the data/size fields, and it is up to the implementation to load the file from disk. The path might be both relative and absolute, and the implementation must check for both cases.

This is useful if the ROM image is too large to load into memory at once. It is also useful if the assests consist of many smaller files, where it is necessary to know the path of a master file to infer the paths of the others. If need_fullpath is set to false, the frontend will load the ROM image into memory beforehand.

In this mode, the path field is not guaranteed to be non-NULL. It should point to a valid path if the file was indeed, loaded from disk, however, it is possible that the file was loaded from stdin, or similar, which has no well-defined path.

It is recommended that need_fullpath is set to false if possible, as it allows more features, such as soft- patching to work correctly.

retro_load_game_special() is a special case of retro_load_game(). It is designed to allow the loading of several ROMs together. This is needed for certain odd cases like Super Nintendo with e.g. Super GameBoy, Sufami Turbo, etc that consist of a "BIOS" + Game(s). The function takes the type of game as an argument, and if a new game type is to be added, it needs to be reserved in the libretro.h header. Almost any libretro implementations should simply implement this as return false;. If a game consist of many smaller files it is encouraged to load a single zipped file, or something similar.

Each ROM image can take an optional meta-argument, a string that gives extra metadeta to the implementation. The metadata is implementation specific, and can be ignored completely in almost any implementation.

4.1.8   retro_get_system_av_info()

This function lets the frontend know essential audio/video properties of the game. As this information can depend on the game being loaded, this info will only be queried after a valid ROM image has been loaded. It is important to accuractely report FPS and audio sampling rates, as FFmpeg recording relies on exact information to be able to run in sync for several hours.

4.2   Running

4.2.1   retro_run()

After a game has been loaded successfully, retro_run() will be called repeatedly as long as the user desires. When called, the implementation will perform its inner functionality for one video frame. During this time, the implementation is free to call callbacks for video frames, audio samples, as well as polling input, and querying current input state. The requirements for the callbacks are that video callback is called exactly once, i.e. it does not have to come last. Also, input polling must be called at least once.

4.2.2   Video/Audio synchronization considerations

Libretro is based on fixed rates. Video FPS and audio sampling rates are always assumed to be constant. Frontends will have control of the speed of playing, typically using VSync to obtain correctspeed. The frontend is free to "fast-forward", i.e. play as fast as possible without waiting, or slow- motion. For this reason, the engine should not rely on system timers to perform arbitrary synchronization. This is common and often needed in 3D games to account for varying frame rates while still maintaining a playable game. However, libretro

targets classic systems where one can assume that 100 % real-time performance will always be met, thus avoiding the need for careful timing code.

By default, the libretro implementation should replace any arbitrary sleep()/time() patterns with simply calling video/audio callbacks. The frontend will make sure to apply the proper synchronization.

This is mostly a problem with game ports, such as PrBoom. For the libretro port of PrBoom, which heavily relied on timers and sleeping patterns, sleeping was replaced with simply running for one frame, and calling the video callback. After that, enough audio was rendered to correspond to one frames worth of time, 1 / fps seconds. All sleeping and timing patterns could be removed, and synchronization was correct.

4.2.3 Audio callback considerations

The libretro API has two different audio callbacks. Only one of these should be used; the implementation must choose which callback is best suited.

The first audio callback is per-sample, and has the type void (*)(int16_t, int16_t). This should be used if the implementation outputs audio on a per-sample basis. The frontend will make sure to partition the audio data into suitable chunks to avoid incurring too much syscall overhead.

If audio is output in a "batch" fashion, i.e. 1 / fps seconds worth of audio data at a time, the batch approach should be considered. Rather than looping over all samples and calling per-sample callback every time, the batch callback should be used instead, size_t (*)(const int16_t *, size_t).

Using the batch callback, audio will not be copied in a temporary buffer, which can buy a slight performance gain. Also, all data will be pushed to audio driver in one go, saving some slight overhead. It is not recommended to use the batch callback for very small ($<$ 32 frames) amounts of data.

The data passed to the batch callback should, if possible, be aligned to 16 bytes (depends on platform), to allow accelerated SIMD operations on audio. RetroArch implements SSE/AltiVec optimized audio processing for conversions and resampling.

4.2.4 Input device abstraction

Abstracting input devices is the hardest part of defining a multi-system API as it differs across every system. The common input devices are:

- Joypad (with or without analogs)
- Mouse (e.g. SNES mouse)
- Keyboard (e.g. Commodore, Amiga)
- Lightguns (e.g. SNES SuperScope)

The joypad abstraction is the most interesting. Rather than complicating things by mapping input arbitrarily in terms of the implementation, which would

make input configuration very complex with careful configuration on a per-implementation basis, an abstract joypad device, the RetroPad, was devised.

This joypad is essentially the Super Nintendo controller, widely considered the pinnacle of retro game controllers. To account for more modern systems with additional buttons, additions from the PlayStation DualShock are incorporated, with extra shoulder buttons (L2/R2), as well as depressable analogs (L3/R3). In addition, the RETRO_DEVICE_ANALOG is used for analog stick data. An implementation should map its idea of a joypad in terms of the RetroPad, which is what most users will have to use with their frontend.

4.2.5   retro_serialize_size()

4.2.6   retro_serialize()

4.2.7   retro_unserialize()

Serialization is optional to implement. Serialization is better known as "save states" in emulators, and these functions are certainly more useful in emulators which have a fixed amount of state. It allows the frontend to take a snapshot of all internal state, and later restore it. This functionality is used to implement e.g. rewind and netplay. Some important considerations must be taken to implement these functions well.

If serialization is not supported, retro_serialize_size() should return 0. If retro_serialize_size() returns non-zero, it is assumed that serialization is properly implemented.

The frontend should call retro_serialize_size() before calling retro_serialize() to determine the amount of memory needed to correctly serialize. The size eventually passed to retro_serialize() must be at least the size of the value returned in retro_serialize_size(). If too large a buffer is passed to retro_serialize(), the extra data should be ignored (or memset to 0).

It is valid for the value returned by retro_serialize_size() to vary over time, however, it cannot ever increase over time. If it should ever change, it must decrease. This is rationaled by the ability to pre- determined a fixed save state size right after retro_load_game() that will always be large enough to hold any following serialization. This certainty is fundamental to the rewind implementation. This requirement only holds between calls to retro_load_game() and retro_unload_game().

If possible, the implementation should attempt to serialize data at consistent offsets in the memory buffer. This will greatly help the rewind implementation in RetroArch to use less memory.

Both retro_serialize() and retro_unserialize() return a boolean value to let the frontend know if the implementation succeeded in serializing or unserializing.

4.3 Tear-down

4.3.1 retro_unload_game()

After the user desired to stop playing, retro_unload_game() will be called. This should free any internal data related to the game, and allow retro_load_game() to be called again.

4.3.2 retro_deinit()

This function should free all state that was initialized during retro_init(). After calling this function, the frontend can again call retro_init().

Index