Implementing a Hardware Accelerated Libretro Core

Hans-Kristian Arntzen

May 9, 2013

Abstract

Libretro API [1] recently received support for cores to use OpenGL (GL2+ or GLES2) directly instead of software rendering to a buffer which is subsequently used by the frontend. This article explains how a core can take advantage of this, and which considerations must be taken into account. This article assumes familiarity with the libretro API.

Cores which use hardware rendering can still use known frontend features, such as multi-pass shaders. This is accomplished by letting cores render to frame buffer objects (FBOs) instead of the back buffer [2].

This addition to the libretro API is designed to be used with hardware-accelerated emulator cores, as well as serving as a framework for graphical demos and experiments.

Application model

Using OpenGL in a libretro context is somewhat different than when you use libraries like SDL, GLFW or SFML. In libretro, the frontend owns the OpenGL context. For an application using conventional libraries like SDL, the application will do this:

- Initialize, create a window of specific size
- Initialize OpenGL resources
- Per frame, handle window events (resize), handle input, render as desired, swap buffers

[1] http://libretro.org

[2] GL drivers must support render-to-texture extensions for this to work.

- Tear down context and window

Using libretro API, platform specifics like managing windows, rendering surfaces and input are all handled by the frontend. The core will only deal with rendering to a surface. The core renders to an FBO of fixed size, determined by the core. The frontend takes this rendered data and stretches to screen as desired by the user. It can apply shaders, change aspect ratio, etc. This model is equivalent to software rendering where `retro_video_refresh_t` callback is called.

Using OpenGL in libretro

- Use `RETRO_ENVIRONMENT_SET_HW_RENDER` environment callback in `retro_load_game()`, notifying frontend that core is using hardware rendering. An OpenGL 2+ or GLES2 context can be specified here. If this is not supported the callback will return false, and you can fallback to software rendering or refuse to start.

- In `retro_get_system_av_info()`, as normal, `max_width` and `max_height` fields specify the maximum resolution the core will render to.

- When the frontend has created a context or reset the context, `retro_hw_context_reset_t` is called. Here, OpenGL resources can be initialized. The frontend can reset the context at will (e.g. when changing from fullscreen to windowed mode and vice versa). The core should take this into account. It will be notified when reinitialization needs to happen.

- A callback to grab OpenGL symbols is exposed via `retro_hw_get_proc_address_t`. Use this to retrieve symbols and extensions.

- In `retro_run()`, use `retro_hw_get_current_framebuffer_t` callback to get which FBO to render to [3]. This is your "backbuffer". Do not attempt to render to the real back buffer. You must call this every frame as it can change every frame. The dimensions of this FBO are at least as big as declared in `max_width` and `max_height`. If desired, the FBO also has a depth buffer attached [4].

[3] e.g. `glBindFramebuffer(GL_FRAMEBUFFER, get_current_framebuffer())`
[4] see `RETRO_ENVIRONMENT_SET_HW_RENDER`

- When done rendering, call `retro_video_refresh_t` with the macro `RETRO_HW_FRAME_BUFFER_VALID` as argument for buffer. Width and height should be specified as well, but pitch argument is irrelevant and will be ignored. If the frame is duped [5], the buffer argument takes `NULL` as normal.

Important considerations in the OpenGL code

The frontend and libretro core share OpenGL context state. Some considerations have to be taken into account for this cooperation to work nicely.

- Don't leave buffers and global objects bound when calling `retro_video_refresh_t`. Make sure to unbind everything, i.e. VAOs, VBOs, shader programs, textures, etc. Failing to do this could potentially hit strange bugs. The frontend will also follow this rule to avoid clashes. Being tidy here is considered good practice anyway.
- The GL viewport will be modified by frontend as well as libretro core. Set this every frame.
- `glEnable()` state like depth testing, etc, is likely to be disabled in frontend as it's just rendering a quad to screen. Enable this per-frame if you use depth testing. There is no need to disable this before calling `retro_video_refresh_t`.
- Avoid VAOs. They tend to break on less-than-stellar drivers [6].
- Try to write code which is GLES2 as well as GL2+ (w/ extensions) compliant. This ensures maximum target surface for the libretro core.

Test implementations

A very basic test implementation of libretro GL interface is available in RetroArch repository on GitHub [7]. It displays two spinning quads. It runs both as a GLES2 and GL2 core depending on `GLES` environment variable.

A slightly more involved test core is found on Bitbucket [8]. It uses instanced rendering of a textured cube, with FPS-style fly-by camera. It uses libretro's mouse API. It is also valid GLES and GL2 at the same time.

[5] RETRO_ENVIRONMENT_CAN_DUPE

[6] At least AMD drivers on Windows are known to break here.

[7] https://github.com/Themaister/RetroArch/tree/master/libretro-test-gl

[8] https://bitbucket.org/Themaister/libretro-gl

Building a libretro core

Libretro is an interface, and not a utility library. Libretro cores are built as standalone dynamic or static libraries, and as they use GL symbols here, they must link against GL symbols themselves.

An example of how this can be done is shown in the test implementation [9].

[9]https://github.com/Themaister/RetroArch/blob/master/libretro-test-gl/Makefile