raysan5 / **raylib_vs_sdl.md**

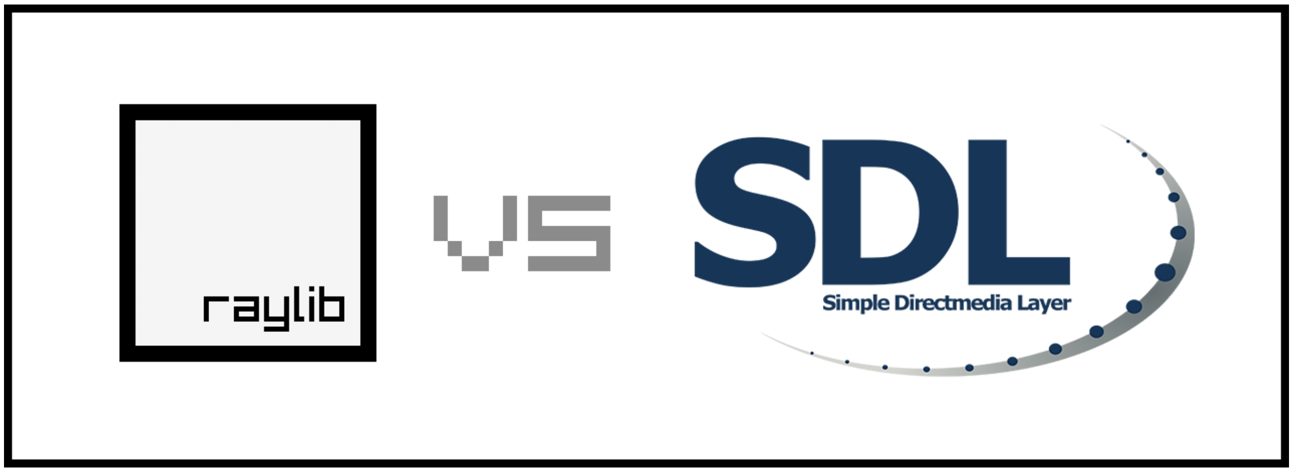
Last active 2 weeks ago • Report abuse

Star

Code Revisions 13 Stars 93 Forks 3

raylib vs SDL - A libraries comparison

raylib_vs_sdl.md



In the last years I've been asked multiple times about the comparison between raylib and SDL libraries. Unfortunately, my experience with SDL was quite limited so I couldn't provide a good comparison. In the last two years I've learned about SDL and used it to teach at University so I feel that now I can provide a good comparison between both.

Hope it helps future users to better understand this two libraries internals and functionality.

Table of Content

Introduction

First time I learned about SDL library was in 2004, by that time I knew very little about programming and I had no idea about videogames development; my first experience with the library was not good. I couldn't understand most of the concepts exposed and how the different parts of the library were connected, SDL naming conventions were confusing to me and the documentation was overwhelming; build configuration was really painful, I remember struggling with the compiler and linkage dependencies, feeling lot of frustration. Since that moment, I developed some apprehension for SDL library.

Eight years later, after having worked in the gamedev AAA industry and also having released several XNA games, I started working as a teacher. I had to teach videogames programming basics to young art students and, at that point, I hardly considered using SDL due to my past experience... instead I decided to create my own library: raylib.

I've been working on raylib for nine years now and a couple of years ago, I started teaching custom game engines development in the University, fate's coincidencies, I was required to teach SDL. I must say that my experience this time was way better than first time, actually, I was able to appreciate this amazing library and I learned a lot from it.

Here it is my personal comparison between raylib and SDL. I tried my best to keep this comparison as much objective as possible but note that my knowledge of SDL internals and its design decisions are more limited than raylib; I could be missing some important piece of information or be confused on some parts.

Please, feel free to comment on this gist with additional information or required improvements.

info	SDL	raylib
Original author	Sam Lantinga (creator and maintainer) Ryan C. Gordon (maintainer)	Ramon Santamaria (creator and maintainer)
Initial release	1998 (24 years ago)	November 2013 (9 years ago)
Latest release	2.26.4 (March 2023)	4.5.0 (March 2023)
Written in	C	C (C99)
Repository	github.com/libsdl-org/SDL	github.com/raysan5/raylib
License	zlib	zlib
Website	libsdl.org	raylib.com

** NOTE: I'm focusing this analysis on SDL 2.0 library, it was a major departure from previous versions, offering more opportunity for 3D hardware acceleration, but breaking backwards-compatibility. License was also changed from GNU LGPL to zlib.*

Library Design Comparison

SDL is similar to raylib in many ways, both are written in C language and offer similar functionality but SDL sits in a bit lower level, providing more grain control over platform/systems configuration options while raylib provides more high-level features.

In terms of platform support, both libraries support multiple platforms: `Windows`, `Linux`, `macOS`, `FreeBSD`, `Android`, `Raspberry Pi`, `HTML5`, `Haiku`... SDL officially supports `iOS` platform while raylib does not but, in any case, both libraries have been ported to additional platforms by the community. SDL has been adapted for multiple console SDKs (under NDA) and raylib has been ported to several homebrew console SDKs.

raylib relies on GLFW library for windowing and input management on Desktop platforms and implements custom windowing/input solutions for other platforms like Android and Raspberry Pi (including native headless mode). Actually, SDL main library provides the same functionality as GLFW (and some more).

Main differences come when looking to library structure, internal conventions and the API provided.

Library Structure

SDL consists of a main library focused on platform/system functionality and some satellite libraries meant to be useful but optional. Users can choose which ones to use depending on the needs of their projects.

SDL library:

- SDL (SDL2.dll): SDL platform/system functionality (window/renderer management, inputs, audio device management, file I/O, timers...)

SDL satellite libraries:

- SDL_image (SDL2_image.dll): Image files loading
- SDL_ttf (SDL2_ttf.dll): Font loading and text rendering (to image)
- SDL_Mixer (SDL2_mixer.dll): Audio files loading and playing
- SDL_net (SDL2_net.dll): Network functionality

raylib is contained into a single library (`raylib.dll` | `libraylib.a`) and desired functionality can be selected with some compilation flags, excluding some of the internal modules or code parts. Internally raylib is divided into a small number of modules (separate .c files), mostly self-contained, providing different functionality:

- core module (rcore.c): Base raylib system functionality (window/graphic device management, inputs, file management, timming...)
- rlgl module (rlgl.h): OpenGL abstraction layer, includes a render batch system. This is a portable single-file header-only library.
- shapes module (rshapes.c): Basic 2D shapes drawing (line, rectangle, circle, poly...)
- textures module (rtextures.h): Image files loading, Image generation and manipulation, Texture2D loading and drawing
- text module (rtext.c): Font loading, font atlas generation, text functionality, text drawing
- models module (rmodels.c): 3D mesh and materials loading, 3D mesh generation, models drawing
- audio module (raudio.c): Audio device management, audio files loading and playing

NOTE: SDL is distributed by default as dynamic libraries while raylib is intended to be linked statically (`libraylib.a`), to avoid external dependencies.

SDL and raylib depend on several external libraries for some functionality (usually file-formats loading). SDL usually relies on the official file-format loading libraries while raylib mostly relies on self-contained single-file header-only libraries that are integrated with the raylib code-base.

`SDL_image` *used to* require the following libraries: `libpng` (+ `zlib`), `libtiff` , `libjpeg` , `libwebp` and `SDL_Mixer` *used to* require `libFLAC` , `libmikmod` , `libogg` , `libvorbis` ... but it seems that latest SDL versions (2022) added support for several single-file header-only libraries as an option (`stb_image` , `dr_flac` , `stb_vorbis` ...), it can be selected at build time.

SDL requires a more complex build system to deal with the many source code files of the library and the multiple external libraries. Here a list of SDL dynamic libraries and it's possible dependencies:

- `SDL2.dll`
- `SDL2_image.dll` could additionally require `libpng16-16.dll` , `libtiff-5.dll` , `libjpeg-9.dll` , `libwebp-4.dll` , `zlib1.dll`
- `SDL2_ttf.dll` could additionally require `libfreetype-6.dll` , `zlib1.dll`

- `SDL2_mixer.dll` could additionally require `libFLAC-8.dll`, `libmodplug-1.dll`, `libogg-0.dll`, `libopus-0.dll`, `libvorbis-0.dll`, `libvorbisfile-3.dll`, `libmpg123-0.dll`
- `SDL2_net.dll`

NOTE: All external libraries can be compiled directly into the SDL dlls to avoid the additional dll requirements, actually, as commented, latest `SDL2_mixer` supports single-file header-only alternatives for several file-formats loading instead of the official ones, it can be configured by the build system.

raylib programs require `raylib.dll` if compiled dynamically or linkage with `libraylib.a` when compiled statically. By default, all required functionality is compiled into this single file. Actually, all required functionality is compiled into only 8 compilation units (8 .c files).

SDL source code is splitted over +700 code files, very well organized by systems, platforms and backends supported while raylib source code consist of a small number of long code files (2000-7000 locs each), at the moment of this writing, raylib consists of 8 individual `.c` files plus some additional portable libraries (about 30 `.h` files).

In terms of library usage, SDL libraries should be included separately, depending on the user needs.

```
#include "SDL.h"            // SDL base library (window/rendered, input, timming...
#include "SDL_opengl.h"     // SDL OpenGL functionality (if required, instead of int

#include "SDL_image.h"      // Image loading functionality (into SDL_Surface)
#include "SDL_ttf.h"        // Font data loading and rasterization (into image)
#include "SDL_mixer.h"      // Audio device management and audio files loading
```

raylib comes with a single header providing all the functionality (intenrally API is divided by sections).

```
#include "raylib.h"         // raylib include (exposes all functionality of every mc
```

Some of the low-level modules included in raylib (used by other higher-level modules) are also available for the users requiring advance features, for example `rlgl.h` and `raymath.h` modules can be included or also used as standalone libraries.

More details about raylib library architecture can be found on raylib Wiki.

Internal Conventions

- **Naming Prefixes**: SDL usually prefixes all functions with `SDL_` (e.g. `SDL_CreateWindow()`) while raylib do not use prefixes on its functions (i.e `InitWindow()`), actually that design decision was motivated by my previous experience with XNA/WinBGI libraries, no-prefix naming was clearer for my students. Worth noticing that `SDL_image` uses the `IMG_` prefix (e.g. `IMG_Load()`), `SDL_ttf` uses the `TTF_` prefix (e.g. `TTF_OpenFont()`) and `SDL_Mixer` uses the `Mix_` prefix (e.g. `Mix_FreeChunk()`). raylib `rlgl` internal module is the only one using the `rl` prefix on all its functions (e.g. `rlLoadRenderBatch()`), motivated by OpenGL convention. A part from the different prefixes decision, both libraries use the `TitleCase` naming convention for its fuctions and structures.

- **Pointers**: SDL uses mostly pointers to opaque structures for most of its data types (e.g. `SDL_Surface`) while raylib uses plain transparent structures that hide the pointers complexity if required. Most SDL functions receive/return those pointers as function parameters so data is passed by reference, raylib instead uses the pass-by-value approach for most of the functions, all the raylib structs have been designed to be as small as possible, usually under 64 bytes.

- **Error Codes**: SDL functions usually return error codes to check if the function worked as expected, raylib usually avoids that approach (only used on some file-access functions). By default, most raylib functions have some fallback mechanism (return a valid default object) and output `LOG_INFO` / `LOG_WARNING` / `LOG_ERROR` messages to console.

- **Backward-compatibility**: SDL is careful with backward-compatibility, old or deprecated functions stay in the library and new ones use different signatures when added. raylib does not care about backward-compatibility; if some function requires to be redesigned or removed, it is. Every new version of raylib list the breaking changes when released but note that they are usually just a few of them.

Functionality Comparison

Both libraries can be used for videogames and graphics apps development, here it is a comparison of the set of functionality provided by both of them.

- File Formats Support
- Window and Renderer System
- Input System

File Formats Support

Let's start with, probably, the less important functionality for professional developers, the input file formats supported to load data into the internal provided structures.

Both libraries support the most common file formats for images, fonts and audio. As commented, SDL allows using the official libraries to load the different file formats (i.e PNG is loaded using `libpng`) while raylib uses single-file header-only alternative libraries to load all the supported file formats (i.e `stb_image` for images).

Here the list of supported file-formats:

type	SDL	raylib
images	BMP, TGA, GIF, JPEG, PNG, QOI, PNM, LBM, PCX, SVG, TIFF, WEBP, XCF, XPM, XV	BMP, TGA, GIF, JPEG, PNG, QOI, PSD, PIC, HDR, DDS, PKM, KTX, PVR, ASTC
audio	WAV, OGG, MP3, FLAC, MOD, XM, S3M, IT, VOC, AIFF...	WAV, OGG, MP3, FLAC, MOD, XM
fonts	TTF, TTC, CFF, WOFF, OTF...	TTF, OTF, FNT, (font atlas images)
meshes	-	OBJ, IQM, GLTF, GLB, VOX

Some points to note:

- raylib supports several GPU compressed file-formats loading but not pixel-data decompression, data is loaded compressed to be moved to a GPU texture directly.
- SDL uses FreeType 2 for fonts rasterization, this is a industry standard library that provides very good rasterization results, supporting several hints like ClearType; raylib instead uses `stb_truetype` library, way smaller than FreeType2 and with a lower font rasterization quality.

- raylib supports 3d meshes/materials/animations loading for several 3d formats, that functionality is not provided by SDL.

Window and Renderer System

> SDL library (`SDL2`) vs raylib core module (`rcore`)

SDL main library exposes +750 functions while raylib core module exposes about 180 functions.

NOTE: Those number include the Input System functionality, that is commented in more detail in the following point.

Both, SDL library and raylib core module provide functions to manage Windows, the graphics device and inputs. SDL implements all the functionality directly using system libraries while raylib relies on GLFW library for desktop platforms (Windows, Linux, macOS...) and implements custom solutions for Android and Raspberry Pi (native mode).

SDL provides a powerful renderer system to draw on the Window, `SDL_Renderer` could be hardware accelerated (usually OpenGL on Linux and Direct3D on Windows) with **software fallback when required**. But, for advance OpenGL features like shaders, it's up to the user to manage it. Actually, it's recommended to avoid the `SDL_Renderer` and request an OpenGL content manually; for that task, another library should be used: `SDL_opengl.h` . `SDL_Renderer` added render batching capabilities on `SDL 2.0.10` .

raylib does not support software rendering and uses **modern OpenGL by default**, it initializes OpenGL 3.3 Core or OpenGL ES 2.0 depending on the platform but, it also supports OpenGL 1.1, 2.1 and 4.3 if desired, thanks to `rlgl` module, an abstraction layer that maps a **pseudo-OpenGL 1.1 immediate mode** to other OpenGL versions. `rlgl` includes a render batching system that accumulates draw calls to issue a single draw when required, batching system is initialized by default on `InitWindow()` but it could be manually managed using some `rlgl` function calls (intended for advance users).

raylib exposes an API for GLSL `Shaders` loading (`LoadShader()`) and usage (`BeginShaderMode()` / `EndShaderMode()`).

Some basic data structures provided by both libraries:

SDL	raylib
`SDL_Rect`	`Rectangle`
`SDL_Color`	`Color`

SDL	raylib
`SDL_Texture`	`Texture2D`
`SDL_Surface`	`Image`

Here it is a comparison of *some* of SDL functions vs raylib equivalent ones, keep in mind that SDL API is bigger than raylib one.

functionality	SDL	raylib
Window/Renderer initialization	`SDL_Init()` `SDL_CreateWindow()` `SDL_CreateRenderer()` `SDL_CreateSoftwareRenderer()` `SDL_CreateWindowAndRenderer()` `SDL_GL_CreateContext()`	`InitWindow()`
Renderer: Clear	`SDL_SetRenderDrawColor()` `SDL_RenderClear()`	`ClearBackground()`
Renderer: Draw basic shapes	`SDL_RenderDrawPoint()` `SDL_RenderFillRect()` `SDL_RenderDrawRect()` `SDL_RenderDrawLine()`	`DrawPixel()` , `DrawCircle()` `DrawRectangle()` `DrawRectangleLines()` `DrawLine()`
Renderer: Draw texture	`SDL_RenderCopy()` `SDL_RenderCopyEx()`	`DrawTexture()` `DrawTextureEx()` `DrawTexturePro()`
Renderer: Blending	`SDL_SetRenderDrawBlendMode()`	`BeginBlendMode()` / `EndBlendMo`
Renderer: Present	`SDL_RenderPresent()` `SDL_GL_SwapWindow()`	`EndDrawing()`
Window/Renderer deinitialization	`SDL_DestroyRenderer()` `SDL_GL_DeleteContext()` `SDL_DestroyWindow()` `SDL_Quit()`	`CloseWindow()`
Timming	`SDL_Delay()`	`WaitTime()`

functionality	SDL	raylib
Logging	`SDL_GetError()` `SDL_Log()`	`TraceLog()`

example 01a: SDL: Window and renderer initialization/deinitialization

```
// Initialize SDL internal global state
// NOTE: SDL allows initializing only desired sub-systems
SDL_Init(SDL_INIT_EVERYTHING);

// Init window
SDL_Window *window = SDL_CreateWindow("Welcome to SDL!", 0, 0, SCREEN_WIDTH, SCR

// Init renderer
// NOTE: SDL_Renderer is not intended for modern OpenGL usage (3d, shaders, adva
// a custom
SDL_Renderer *renderer = SDL_CreateRenderer(state.window, -1, SDL_RENDERER_ACCEL
SDL_SetRenderDrawColor(renderer, 100, 149, 237, 255);  // Default clear color: C

// App loop
while (!closeWindow)
{
    // Process input events

    // Draw
}

// Deinitialize renderer and window
SDL_DestroyRenderer(renderer);
SDL_DestroyWindow(window);

// Deinitialize SDL internal global state
SDL_Quit();
```

Note that `SDL_Renderer` is not intended for modern OpenGL capabilities (3d, shaders, advance graphic features...), an OpenGL context should be manually created for that case.

example 01b: SDL: Window and renderer initialization/deinitialization (modern OpenGL)

```
// OpenGL functionality required
#include "SDL_opengl.h"
```

```
// Initialize SDL internal global state
SDL_Init(SDL_INIT_EVERYTHING);

// Init window
SDL_Window *window = SDL_CreateWindow("Welcome to SDL OpenGL!", 0, 0, SCREEN_WID

// Init OpenGL context
SDL_GLContext glcontext = SDL_GL_CreateContext(window);

// App loop
while (!closeWindow)
{
    // Process input events

    // Draw using OpenGL
    // NOTE: It's up to the user to load required extensions and manage OpenGL s
    glClearColor(0,0,0,1);
    glClear(GL_COLOR_BUFFER_BIT);
    SDL_GL_SwapWindow(window);
}

// Deinitialize OpenGL context and window
SDL_GL_DeleteContext(glcontext);

// Deinitialize SDL internal global state
SDL_Quit();
```

example 01c: raylib: Window and renderer initialization/deinitialization (modern OpenGL)

```
// Init raylib internal global state and create window
InitWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Welcome to raylib!");

// App loop...
while (!WindowShouldClose())
{
    // Draw
    BeginDrawing();
        ClearBackground(BLACK);

    // Input events are processed internally by default
    EndDrawing();
}
```

```
    // Deinitialize raylib internal global state and window
    CloseWindow();
```

Input System

SDL exposes +100 functions to deal with multiple input devices while raylib exposes about 45 functions to deal with inputs.

SDL provides a mechanism to poll input events and provide that "raw" data directly to the user. raylib instead polls inputs internally by default on the `EndDrawing()` call (advance users can do the polling by themself calling `PollInputEvents()`) and fills internal buffers with result data to provide some high-level functions to check inputs state (i.e `IsKeyPressed()` , `IsGamepadButtonDown()` ...).

Both libraries supports the standard input systems: Keyboard, Mouse, Gamepad, Touch events and Gestures. SDL also includes functionality to manage Sensors and Haptic controls.

Some points to note:

- SDL gamepad support is outstanding. It provides really fine-grain control of gamepad data and it supports gamepad mappings for multiple devices. Actually, GLFW uses the same `SDL_GameControllerDB` data, so, it is also used internally by raylib to indentify many gamepad controllers.

functionality	SDL	raylib
Input events polling	`SDL_PollEvents()` `SDL_PumpEvents()`	`PollInputEvents()` (called by `EndDrawing()`)
Keyboard inputs	`SDL_PollEvent()` (check `event.type`) `SDL_GetKeyboardState()`	`IsKeyPressed()` , `IsKeyDown()` , `GetKeyPressed()`
Mouse inputs	`SDL_PollEvent()` (check `event.type`) `SDL_GetMouseState()`	`IsMouseButtonPressed()` , `IsMouseButtonDown()` , `GetMousePosition()`
Gamepad inputs	`SDL_PollEvents()` (check `event.type`) `SDL_NumJoysticks()`	`IsGamepadAvailable()` `IsGamepadButtonPressed()` `IsGamepadButtonDown()` `GetGamepadAxisMovement()`

functionality	SDL	raylib
	`SDL_JoystickOpen()` `SDL_JoystickClose()`	
Input constants (sample)	`SDL_SCANCODE_ESCAPE` `SDL_SCANCODE_UP` `SDL_SCANCODE_DOWN` `SDL_SCANCODE_LEFT` `SDL_SCANCODE_RIGHT` `SDL_SCANCODE_SPACE`	`KEY_ESCAPE` `KEY_UP` `KEY_DOWN` `KEY_LEFT` `KEY_RIGHT` `KEY_SPACE`

example 02a: SDL: Input management

```
    // Init input gamepad
    // Check SDL_NumJoysticks() and SDL_JoystickOpen()
    if (SDL_NumJoysticks() < 1) SDL_Log("WARNING: No joysticks connected!\n");
    else
    {
        SDL_Joystick *gamepad = SDL_JoystickOpen(0);
        if (SDL_Joystick *gamepad == NULL) SDL_Log("WARNING: Unable to open game cor
    }

    // Poll input events
    SDL_Event event = { 0 };
    SDL_PollEvent(&event);

    // All input events can be processed after polling
    switch(event.type)
    {
        case SDL_QUIT: closeWindow = true; break;

        // Window events are also polled (Minimized, maximized, close...)
        case SDL_WINDOWEVENT: break;

        // Keyboard events
        case SDL_KEYDOWN:
        {
            if (event.key.keysym.sym == SDLK_ESCAPE) closeWindow = true;
        } break;

        // Check mouse events
        case SDL_MOUSEBUTTONDOWN: break;
        case SDL_MOUSEBUTTONUP: break;
        case SDL_MOUSEWHEEL: break;
        case SDL_MOUSEMOTION:
        {
            mousePosition.x = event.motion.x;
            mousePosition.y = event.motion.y;
        } break;

        // Check gamepad events
        case SDL_JOYAXISMOTION:
        {
            // Motion on gamepad 0
            if (event.jaxis.which == 0)
            {
                // X axis motion
                if (event.jaxis.axis == 0)
                {
                    //...
                }
                // Y axis motion
                else if (event.jaxis.axis == 1)
                {
                    //...
                }
            }
        } break;
        default: break;
    }
```

example 02b: raylib: Input management

```
        // Inputs polling is done at EndDrawing() by default and states are managed inte

        // Check if one key has been pressed (UP->DOWN)
        if (IsKeyPressed(KEY_ENTER)) currentScreen = SCREEN_GAMEPLAY;

        // Check if one key is being pressed (DOWN)
        if (IsKeyDown(KEY_LEFT)) player.position.x -= player.speed.x;
        else if (IsKeyDown(KEY_RIGHT)) player.position.x += player.speed.x;

        // Check if mouse button has been pressed and get mouse position
        if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT)
        {
            Vector2 mousePosition = GetMousePosition();
        }

        // Check gamepad button pressed
        if (IsGamepadButtonPressed(GAMEPAD_BUTTON_LEFT_FACE_RIGHT)) { }
```

```
    // Get gamepad axis movement
    float axisValue = GetGamepadAxisMovement(0, GAMEPAD_AXIS_RIGHT_TRIGGER);
```

Shapes Drawing

> SDL library (SDL2) vs raylib shapes module (rshapes)

NOTE: SDL provides those functions as part of the main library while raylib provides this functionality contained in the `rshapes` module.

SDL exposes several functions to render Points, Lines and Rectangles while raylib provides about 40 functions to render multiple types of shapes including `Circle` , `Ellipsis` , `Ring` , `Triangle` and more.

SDL requires setting the blending and render color using the functions `SDL_SetRenderDrawBlendMode()` and `SDL_SetRenderDrawColor()` . raylib uses a default blending mode with alpha and all `Draw*()` functions expect a last `Color` parameter for coloring/tinting by default.

raylib also defines some *custom* colors for convenience (`RED` , `BLUE` , `DARKGREEN` , `RAYWHITE` ...).

functionality	SDL	raylib
Point drawing	`SDL_RenderDrawPoint*()`	`DrawPixel()`
Line drawing	`SDL_RenderDrawLine*()`	`DrawLine*()`
Rectangle drawing	`SDL_RenderFillRect*()` `SDL_RenderDrawRect*()`	`DrawRectangle*()` `DrawRectangleLines*()`
Arbitrary shapes	`SDL_RenderGeometry()`	`DrawTriangle*()` `DrawPoly*()`

example 03a: SDL: Draw rectangle shape

```
    // Set blending and color for shapes drawing
    SDL_SetRenderDrawBlendMode(renderer, SDL_BLENDMODE_BLEND);
    SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);   // RED

    // Draw a red rectangle at position (10, 10) with size (200, 180)
    SDL_Rect rec = { 10, 10, width, height };
    SDL_RenderFillRect(renderer, &rec);
```

example 03b: raylib: Draw rectangle shape

```
    // Draw a red rectangle at position (10, 10) with size (200, 180)
    DrawRectangle(10, 10, 200, 180, RED);
```

Images and Textures Loading and Drawing

> SDL library (`SDL_Image`) vs raylib textures module (`rtextures`)

SDL loads image data into `SDL_Surface` opaque structures while raylib uses `Image` structure.

functionality	SDL	raylib
Image system init/deinit	`IMG_Init()` `IMG_Quit()`	*not required*
Image loading	`IMG_Load()`	`LoadImage()`
Load texture from image	`SDL_CreateTextureFromSurface()`	`LoadTexture()` `LoadTextureFromImage()`
Unload image and texture	`SDL_FreeSurface()` `SDL_DestroyTexture()`	`UnloadImage()` `UnloadTexture()`
Draw texture	`SDL_RenderCopy()` `SDL_RenderCopyEx()`	`DrawTexture()` `DrawTextureEx()` `DrawTexturePro()`
Update texture data	`SDL_UpdateTexture()`	`UpdateTexture()`
Read texture pixel data	`SDL_RenderReadPixels()`	`LoadImageFromTexture()`

One difference with SDL is that raylib provides multiple functions for image generation and manipulation. For example, it includes functions to generate `gradient` , `checked` , `white noise` or `cellular` images and also functions to `format` , `resize` , `rotate` . `crop` , `colorize` and more image operations.

SDL supports software renderering, raylib does not have that option but it exposes multiple functions to **draw on images**, that it is actually 2d software rendering capabilities, including text drawing.

Both libraries support multiple pixel-formats: SDL and raylib.

example 04a: SDL: Image/Texture loading

```
// Image loading functionality required
#include "SDL_image.h"

// Init image system
IMG_Init(IMG_INIT_PNG);

// Load image and load texture from it,
// note that returned types are actually opaque pointers
SDL_Surface *image = IMG_Load("sample.png");
SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, image);
SDL_FreeSurface(surface);   // Image can be unloaded once loaded to texture

// Texture properties must be queried due to the opaque pointers
int format, access, width, height;
SDL_QueryTexture(texture, &format, &access, &width, &height);
```

example 04b: raylib: Image/Texture loading (long option):

```
// Load image and load texture from that image,
// note that returned types are transparent structs
Image image = LoadImage("sample.png");
Texture2D texture = LoadTextureFromImage(image);
UnloadImage(image);     // Image can be unloaded once loaded to texture

// Texture data can be directly accessed: texture.width, texture.height, texture
```

example 04c: raylib: Image/Texture loading (short option):

```
// Texture can be directly loaded from image file,
// actually, the internal process is the same than previous example
Texture2D texture = LoadTexture("sample.png");
```

Render to Texture Support

Both libraries support offline rendering to texture, SDL allows creating specific `SDL_Texture` while raylib provides `RenderTexture` structure.

functionality	SDL	raylib
Data structure	`SDL_Texture`	`RenderTexture`
Load render texture	`SDL_CreateTexture`	`LoadRenderTexture()`
Unload render texture	`SDL_DestroyTexture()`	`UnloadRenderTexture()`
Enable render texture (for drawing)	`SDL_SetRenderTarget()`	`BeginTextureMode()` `EndTextureMode()`

example 04d: SDL: Load and draw to render texture

```
// Init texture for offline rendering
SDL_Texture *texture = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_RGBA8888, SDL

// Enable render texture for drawing
SDL_SetRenderTarget(renderer, texture);

// Draw a point into the texture (setting blending and color)
SDL_SetRenderDrawBlendMode(renderer, SDL_BLENDMODE_BLEND);
SDL_SetRenderDrawColor(renderer, 255, 0, 255, 255);
SDL_RenderDrawPoint(renderer, 123, 456);

// Enable window to continue drawing
SDL_SetRenderTarget(renderer, NULL);
```

example 04e: raylib: Load and draw to render texture

```
// Load render texture for offline rendering
// NOTE: By default a 32bit RGBA color buffer and depth renderbuffer is created
RenderTexture target = LoadRenderTexture(1024, 1024);

// Enable render texture for drawing
BeginTextureMode(target);

    // Draw a point into the texture
    DrawPixel(123, 456, MAGENTA);
// Enable window to continue drawing
EndTextureMode();
```

Font Loading and Text Drawing

> SDL library (`SDL_ttf`) vs raylib text module (`rtext`)

SDL uses `FreeType2` and `Harfbuzz` libraries internally, two powerful libraries providing high-quality text rasterization options, raylib instead relies on `stb_truetype` library for font rasterization, with lower-quality results.

The text rendering approach of SDL and raylib is a bit different, `SDL_ttf` exposes several convenience functions (`TTF_RenderText*()`) to render text into a `SDL_Surface` that later on can be used on SDL drawing. raylib `rtext` module uses `rtextures` and `rlgl` functionality; it supports automatic font atlas generation and text drawing using the internal `rlgl` batching system.

raylib `rtext` module also provides several functions to manage codepoint and UTF-8 text and some general-pourpose string management functions for convenience.

raylib includes a default font embedded, initialized at `InitWindow()` , so users can call `DrawText()` by default, with no worries about loading a font previously.

example 05a: SDL: Font loading and text drawing:

```
    // Font loading functionality required
    #include <SDL_ttf.h>

    // Initialize fonts system
    TTF_Init();

    // Load font from file
    TTF_Font *font = TTF_OpenFont("arial.ttf", 24);

    // Render text into a surface using provided font and color
    SDL_Color color = { 255, 255, 255 };
    SDL_Surface *surface = TTF_RenderText_Solid(font, "Welcome to SDL", color);

    // Create texture from surface
    SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, surface);

    // Draw texture (containing text)
    int texWidth = 0, texHeight = 0;
    SDL_QueryTexture(texture, NULL, NULL, &texWidth, &texHeight);
    SDL_Rect dstrect = { 0, 0, texWidth, texHeight };

    SDL_RenderCopy(renderer, texture, NULL, &dstrect);
    SDL_RenderPresent(renderer);

    // Unload texture (containing text)
    SDL_DestroyTexture(texture);
    SDL_FreeSurface(surface);

    // Unload font and quit font system
    TTF_CloseFont(font);
    TTF_Quit();
```

example 05b: raylib: Text drawing (default internal font):

```
    // Draw text on the screen at position (100, 200), 20 pixels height, RED color
    DrawText("Welcome to raylib", 100, 200, 20, RED);
```

example 05c: raylib: Font loading and text drawing (custom font):

```
    // Load font from file
    // NOTE: raylib generate a font atlas automatically with some default parameters
    // LoadFontEx() is also provided for a more fine control over font size and requ
    Font font = LoadFont("arial.ttf");

    // Draw text on screen using loaded font, internal batching system makes this pr
    DrawTextEx(font, "Welcome to raylib", (Vector2){ 10.0f, 10.0f }, 20, 1, RED);

    // Unload font
    UnloadFont(font);
```

Models Loading and Drawing

> *none* vs raylib models module (`rmodels`)

raylib provides a module focused on 3d rendering. This is a high-level feature not include in SDL.

This module includes functions to draw basic 3d shapes (cube, sphere, cone...), functions to generate 3d meshes (cube, cillinder, heightmap...), functions to load meshes/materials/anims from 3d files (.obj, .iqm, .gltf, .vox) and some functions to check basic 3d collisions (Ray, BoundingBox).

Audio System

> SDL library (`SDL2` and `SDL_Mixer`) vs raylib audio module (`raudio`)

Both libraries provide similar functionality, they allow loading and playing static and dynamic (streaming) audio data and manage the audio device. SDL main library provides the audio device management functionality (initialized by `SDL_Init()` or `SDL_InitSubSystem()`) while `SDL_Mixer` provides multiple audio file-formats loading and playing functionality.

raylib audio module contains device management functionality and audio files loading and playing. For audio device management, it relies on the single-file header-only `miniaudio` library, supportting multiple audio backends and some advance features.

SDL provides support for audio effects while raylib added an *experimental* first approach to real-time effects in latest `raylib 4.2` release.

functionality	SDL	raylib
Init audio system	`Mix_Init()` `Mix_OpenAudio()`	`InitAudioDevice()`
Close audio system	`Mix_CloseAudio()` `Mix_Quit()`	`CloseAudioDevice()`
Load/unload sound	`Mix_LoadWAV()` `Mix_FreeChunk()`	`LoadSound()` `UnloadSound()`
Load/unload music	`Mix_LoadMUS()` `Mix_FreeMusic()`	`LoadMusicStream()` `UnloadMusicStream()`
Play sound	`Mix_PlayChannel()`	`PlaySound()`
Play music	`Mix_PlayMusic()`	`PlayMusicStream()` `UpdateMusicStream()`
Sound data struct	`Mix_Chunk`	`Sound`
Music data struct	`Mix_Music`	`Music`

example 06a: SDL: Audio loading and playing

```
#include "SDL_mixer.h"      // Audio device management and audio files loading r

// Init audio system
Mix_Init(MIX_INIT_OGG);
Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048);

// Load sound file (full audio data loaded into memory)
Mix_Chunk *sound = Mix_LoadWAV("Assets/sound.wav");

// Load music file (only one data piece, intended for streaming)
Mix_Music *music = Mix_LoadMUS("Assets/music.ogg");

// Play sound file once
Mix_PlayChannel(-1, sound, 0);

// Start playing streamed music
Mix_PlayMusic(music, -1);

// Unload sound and music data
Mix_FreeMusic(music);
Mix_FreeChunk(sound);

// Close audio system
Mix_CloseAudio();
Mix_Quit();
```

example 06b: raylib: Audio loading and playing

```
// Initialize audio device (default config: 44100Hz, 2 channels)
InitAudioDevice();

// Load sound file (full audio data loaded into memory)
Sound sound = LoadSound("Assets/sound.wav");

// Load music file (only one data piece, intended for streaming)
Music music = LoadMusicStream("Assets/music.ogg");

// Play sound file once
PlaySound(sound);

// Start playing streamed music
PlayMusicStream(music);

// Refill music stream buffers if already processed (required to be called by us
UpdateMusicStream(music);

// Unload sound and music data
UnloadMusic(music);
UnloadSound(sound);

// Close audio device
CloseAudioDevice();
```

Network System

> SDL library (`SDL_net.dll`) vs *none*

SDL includes a module for networking while raylib has not networking support.

Worth mentioning that at some point `rnet` module was available but it was buggy and unmaintained so it was finally removed.

Conclusions

SDL provides a more fine-grained control over the platform than raylib. raylib provides some higher-level functionality and takes multiple decisions for the user to simplify API usage.

SDL has been around for +25 years and it's a batttle-tested library, used in many professional products. raylib is about 9 years old and it was originally created for education, still today most of its user-base are students and hobbyist.

Fortunately in the future, this article will help users to decide which one of the two libraries better fit their products needs.

Thanks for reading and, please, feel free to comment on this gist with additional information or required improvements. :)

bferguson3 commented on Aug 17, 2022

Did you do any performance tests? Curious to see results

raysan5 commented on Aug 17, 2022 • edited Author