SDL3
THE COMPLETE REFERENCE

A COMPREHENSIVE GUIDE TO CROSS-PLATFORM GAME DEVELOPMENT WITH SIMPLE DIRECTMEDIA LAYER 3

- GRAPHICS
- AUDIO
- INPUT
- PLATFORM
- ENGINEERING

MIKEL SOKOLIUK

SDL3: The Complete Reference

A Comprehensive Guide to Cross-Platform Game Development with Simple DirectMedia Layer 3

Steve Publications

This book is available at https://leanpub.com/sdl3thecompletereference

This version was published on 2026-07-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Comprehensive Guide to Cross-Platform Game Development with Simple DirectMedia Layer 3

About this book

SDL3 is the modern, production-ready evolution of the legendary cross-platform development library used by Valve games, emulators, and millions of applications worldwide. This book takes you from your first SDL3 window through advanced GPU rendering, audio engineering, input management, and everything in between. Every chapter contains complete, compilable code examples that build progressively from simple programs to reusable architectures. Whether you are migrating an SDL2 project or starting fresh with SDL3, this book is designed to be the single reference you keep on your desk.

Introduction: Why SDL3?

The Legacy of Simple DirectMedia Layer

In early 1998, Sam Lantinga released the first version of what would become one of the most influential libraries in game development history [1,43]. Simple DirectMedia Layer, or SDL, was born from a practical need: Lantinga, then working at Loki Software (a company specializing in porting Windows games to Linux), wanted a consistent, cross-platform way for C and C++ programmers to talk to audio, keyboard, mouse, joystick, and graphics hardware without writing platform-specific code for every operating system [1,43].

The result was transformative. SDL became the foundation for thousands of games, emulators, media players, and tools. By 2010, over 700 games and applications were using SDL [43]. Valve's Source Engine relied on SDL for input handling and window management across its award-winning catalog, including *Half-Life* 2, *Portal*, and *Counter-Strike: Source* [44,43]. The RetroArch emulator frontend chose SDL as its primary input and display layer. FFmpeg used it for video playback windows. Godot Engine built its windowing system on top of it. Countless indie developers shipped their first games using nothing more than SDL and a text editor.

By the time SDL 2.0 reached stability in August 2013 (after an initial announcement in 2012), the library had evolved into something far larger than its name suggested [43]. It supported OpenGL, OpenGLES, Direct3D, Vulkan (added later), Metal, and software rendering. It managed audio on every major platform. It provided game controller standardization that made supporting dozens of different controllers trivial. And it did all this with an API so simple that a beginner could create a window and draw a triangle in under fifty lines of code.

What Changed in SDL3

SDL3 is not a minor version bump. It is a ground-up redesign that addresses two decades of accumulated technical debt while introducing capabilities that

simply did not exist in the SDL2 era. The first official stable release, SDL 3.2.0, arrived on January 21, 2025, and the ABI was locked at that point, meaning the API you learn today will remain stable for years to come [40,41,43]. (The version number starts at 3.2.0 rather than 3.0.0 because the SDL team used pre-release versions numbered 3.0.x and 3.1.x during development, and the first "stable" release inherited the next version in sequence.)

The changes are substantial:

- **A new GPU API** provides cross-platform access to modern graphics hardware (Vulkan, Metal, Direct3D 12) with a unified interface [17,53]. Write shaders once, deploy everywhere. This is the kind of abstraction that previously required Unity or Unreal Engine, or your own custom cross-compilation pipeline.
- **Redesigned audio** replaces the callback-based model with a stream architecture (`SDL_AudioStream`) that handles format conversion, mixing, and buffering more elegantly [14,15]. Multiple audio sources mix automatically, and you can bind as many streams to a device as you need. SDL also introduced logical vs. physical device separation, allowing multiple independent audio connections to share one hardware device [52].
- **Restructured events** use nanosecond timestamps (`Uint64` instead of `Uint32` milliseconds), float mouse coordinates (sub-pixel precision), and a new `SDL_EVENT_*` naming convention [8,9]. The event system uses a mutex-protected global queue capped at 65,535 events [50].
- **Properties API** replaces dozens of scattered getter/setter functions with a unified key-value system (`SDL_PropertiesID`) for windows, renderers, textures, gamepads, and other objects [23]. This makes the API extensible without breaking changes.
- **New subsystems** for camera access, file dialogs, storage management, pen input, process spawning, and application metadata bring SDL3 closer to a complete application framework [40].
- **Main callbacks** (`SDL_AppInit`, `SDL_AppEvent`, `SDL_AppIterate`, `SDL_AppQuit`) offer an alternative to the traditional main loop, making it trivial to write code that works on iOS, Android, Emscripten, and desktop without platform-specific boilerplate [33,36].
- **Breaking API changes** mean SDL2 code will not compile against SDL3 without modification [38]. This is intentional: the SDL3 team chose to fix long-standing design issues rather than perpetuate them. Automated migration scripts (Python-based rename tools) make the transition manageable [38].

The library name changed from `SDL2` to `SDL3`, the include path changed from `<SDL2/SDL.h>` to `<SDL3/SDL.h>`, and most functions now return `bool` instead of negative error codes. These changes ensure that SDL2 and SDL3 can coexist on the same system without conflict.

Who Should Read This Book

This book is written for C and C++ developers who want to build cross-platform graphical applications, games, emulators, or multimedia tools. You should be comfortable with basic programming concepts: variables, functions, pointers, structs, and memory management. If you have used SDL2 before, you will find the transition straightforward thanks to the dedicated migration chapter. If you are new to SDL entirely, this book starts from zero and builds up systematically.

You do not need to be a graphics programmer, audio engineer, or systems expert. The book explains concepts as it introduces them. When we discuss Vulkan integration, for example, the focus is on how SDL3 manages the platform-specific details so you can concentrate on your application logic rather than driver quirks.

SDL3 Architecture at a Glance

Before diving into specific APIs, it helps to understand how SDL3 is structured internally. This overview explains the library's modular architecture, thread model, and the design rationale behind its major subsystems. You can skim this section now and return to it as context for later chapters.

Modular Subsystem Design

SDL3 is organized into independent subsystems that can be initialized separately. When you call `SDL_Init(SDL_INIT_VIDEO)`, only the video subsystem starts. The audio, joystick, haptic, sensor, and timer subsystems remain dormant until you explicitly initialize them [2,3]. Each subsystem:

- Has its own initialization and shutdown functions
- Maintains independent internal state
- Can be initialized from any thread (though most subsystems require main-thread usage for their public APIs)

- Contributes to the global event queue when relevant

This modular design means your application only pays the cost of the subsystems it actually uses. A simple text-based tool that only needs keyboard input and a window does not initialize audio, gamepad, or sensor drivers.

The Thread Model

SDL3 follows a strict threading discipline:

Must Be Main Thread	Can Be Any Thread	Dedicated SDL Thread
`SDL_Init`, `SDL_Quit`	`SDL_GetError`	Audio processing (one per physical device)
Window creation/destruction	Logging functions	Event pumping (`SDL_PumpEvents`)
Renderer operations	Atomic operations	—
OpenGL/Vulkan/Metal context calls	Properties API (thread-safe)	—
Event polling (`SDL_PollEvent`)	Hint setting/getting	—

The video and rendering subsystems are strictly single-threaded [51]. All window creation, renderer operations, and graphics API context calls must happen on the main thread. The audio subsystem runs its own processing thread per physical device, mixing bound streams and delivering data to the OS audio driver [52]. The event queue is protected by a mutex, allowing safe addition from multiple threads, though polling should happen on the main thread for consistency.

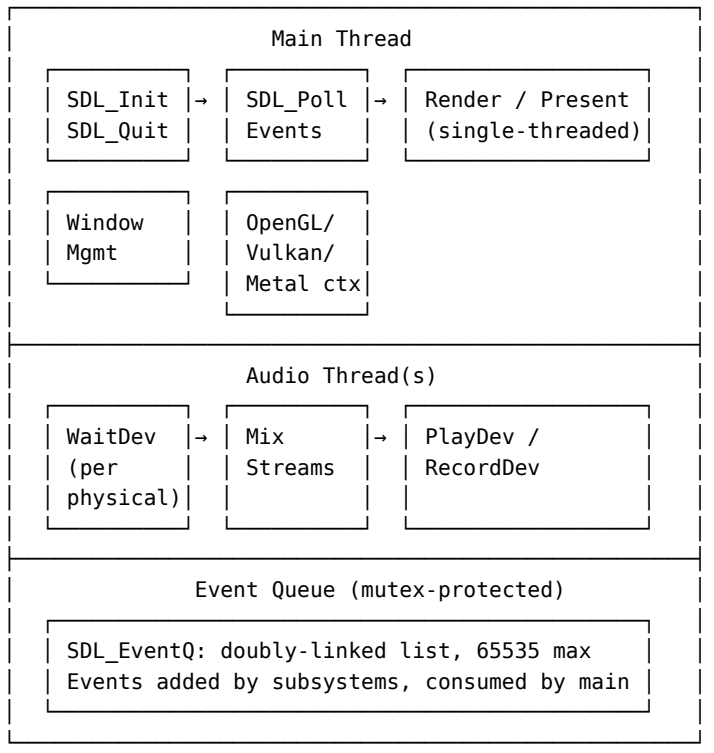

Figure: SDL3 thread model showing the main thread responsible for video/rendering/event polling, dedicated audio threads per physical device, and the shared mutex-protected event queue.

Backend Abstraction Layer

One of SDL3's core design principles is backend abstraction. For rendering, multiple drivers exist (Direct3D 11/12, Metal, Vulkan, OpenGL, software) [51]. For audio, platform-specific drivers handle communication with the OS (CoreAudio on macOS, WASAPI on Windows, PulseAudio/PipeWire on Linux) [52]. The application code never touches these backends directly; SDL3 translates your high-level API calls into the appropriate backend operations.

The GPU API takes this further by abstracting not just window management but the entire 3D rendering pipeline. A single `SDL_GPUDevice` can run on Vulkan, Metal, or Direct3D 12 depending on what is available [17,53]. Shaders are compiled to the native format of each backend, and SDL provides the `SDL_shadercross` tool for HLSL-to-SPIR-V/MSL cross-compilation [17].

The Initialization Lifecycle

Every SDL3 application follows this lifecycle:

1. **`SDL_Init(subsystem_flags)`**: Starts requested subsystems. Returns `bool` (true on success).
2. **Application setup**: Create windows, renderers, load assets, initialize game state.
3. **Main loop**: Process events, update game logic, render frames.
4. **`SDL_Quit()`**: Shuts down all initialized subsystems in reverse order of initialization. Destroys property groups and frees global resources [23].

SDL3's `bool` return values (instead of SDL2's integer error codes) simplify error checking: every function that can fail returns true on success and false on failure, with details available via `SDL_GetError()` [30].

Design Trade-offs Worth Understanding

Decision	Rationale	Trade-off
Breaking API from SDL2	Fix accumulated design debt; bool returns, stream audio, event naming	Existing SDL2 code requires migration; no automatic compatibility layer
Stream-based audio	Automatic mixing, format conversion, multiple sources per device	Steeper learning curve for SDL2 users accustomed to callbacks
Deferred renderer (command queue)	Automatic batching reduces GPU driver overhead	Must flush before direct GL/Vulkan calls; less fine-grained control
Properties API over getters/setters	Extensible without breaking changes; uniform across objects	Slightly more verbose than a single getter function
Single-threaded rendering	Simpler internal state management; no renderer mutex contention	Cannot record GPU commands from multiple threads (use SDL_GPU for that)

These trade-offs reflect the SDL team's prioritization of simplicity, cross-platform consistency, and forward compatibility over backward compatibility and maximum low-level control. For most applications, these choices result in less code, fewer bugs, and easier maintenance.

How to Use This Book

The chapters are organized to take you from first principles to advanced mastery:

- **Chapters 1 through 3** cover installation, window creation, events, and the built-in 2D renderer. By the end of Chapter 3, you will be able to create a complete graphical application with images, text, and user input.
- **Chapters 4 through 7** dive into surfaces, audio, fonts, and input devices. These chapters give you the tools to build feature-rich applications with sound, text rendering, gamepad support, and haptic feedback.
- **Chapters 8 through 10** tackle threading, file I/O, and direct graphics API integration (OpenGL, Vulkan, Metal, and the new GPU API). This is where you learn to build high-performance, production-quality applications.
- **Chapters 11 and 12** cover advanced platform features and the SDL2-to-SDL3 migration path.

Each chapter contains complete, compilable code examples. You can copy them directly into your project, compile them, and run them.

The PixelQuest Mini-Project. In addition to standalone examples, this book builds a running mini-project called **PixelQuest** – a simple 2D platformer game. Each chapter adds new subsystems to the growing codebase:

- Chapter 1 creates the window and basic event loop
- Chapter 2 adds keyboard and mouse input handling
- Chapter 3 renders the player sprite and background tiles
- Chapter 4 uses surfaces for pixel-level sprite effects
- Chapter 5 adds sound effects and background music
- Chapter 6 displays score, health, and menu text
- Chapter 7 adds gamepad support for console-style controls
- Chapter 8 introduces a worker thread for physics updates
- Chapter 9 loads level data from configuration files

- Chapter 10 explores an optional GPU API rendering path

At the end of each chapter, you will find a "PixelQuest Integration" section showing exactly how to plug the new material into the evolving project. By Chapter 10, you will have a complete, compilable game skeleton with multiple subsystems working together.

Code examples use modern C (C11 or later) and modern C++ (C++17 or later) where appropriate. Compilation instructions accompany every example. Error handling is included in every example, because production code always handles errors.

Throughout the book, you will encounter callout boxes that highlight common pitfalls, performance tips, and design trade-offs. These are drawn from real-world experience building SDL applications and are meant to save you hours of debugging.

Let us begin.

Chapter 1: Getting Started with SDL3

Installing SDL3 on Windows

On Windows, there are several ways to install SDL3 depending on your preferred toolchain. The most common approaches use MSYS2/MinGW, vcpkg, or building from source with Visual Studio or CMake.

Using MSYS2 and MinGW-w64

MSYS2 provides a Unix-like development environment on Windows with the pacman package manager. It is one of the most straightforward ways to get SDL3 up and running:

1. Download and install MSYS2 from https://www.msys2.org/
2. Open the MSYS2 UCRT64 terminal (or MINGW64 if you prefer that environment)
3. Update the package database: `pacman -Syu`
4. Install development tools: `pacman -S base-devel mingw-w64-ucrt-x86_64-gcc mingw-w64-ucrt-x86_64-cmake`
5. Install SDL3: `pacman -S mingw-w64-ucrt-x86_64-SDL3`

After installation, the headers are available at `/ucrt64/include/SDL3/` and the library files at `/ucrt64/lib/`. You can compile a simple program with:

```
gcc main.c -o main.exe $(pkg-config --cflags --libs sdl3)
```

Note: MSYS2's SDL3 package tracks the latest release. As of this writing, the package provides SDL 3.2.x or later.


```
gcc main.c -o main.exe $(pkg-config --cflags --libs sdl3)
```


Using vcpkg

Microsoft's vcpkg package manager can install SDL3 for Visual Studio or MinGW:

```
vcpkg install sdl3:x64-windows
```

This installs SDL3 as a CMake package. Your CMakeLists.txt can then use `find_package(SDL3 REQUIRED)` to locate it.

Using NuGet (Visual Studio)

If you are using Visual Studio with the .NET build system, SDL3 is available as a NuGet package:

```
Install-Package SDL3
```

This automatically configures include directories and library linking in your Visual Studio project.

Installing SDL3 on macOS

On macOS, Homebrew is the recommended installation method:

```
brew install sdl3
```

This installs SDL3 as a framework in `/opt/homebrew/opt/sdl3/` (Apple Silicon) or `/usr/local/opt/sdl3/` (Intel). The CMake config files are installed automatically, so `find_package(SDL3 REQUIRED)` works out of the box.

For Xcode projects, you can use CocoaPods:

```
pod 'SDL3', '~> 3.2'
```

Or manually add the SDL3 framework to your Xcode project by dragging it from `/Library/Frameworks/` (or the Homebrew prefix) into your project navigator.

Building from source on macOS requires Xcode command line tools:

```
xcode-select --install
git clone https://github.com/libsdl-org/SDL.git
cd SDL
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build
sudo cmake --install build
```

Installing SDL3 on Linux

Linux distribution package managers provide SDL3 for most major distros:

Debian/Ubuntu:

```
sudo apt install libsdl3-dev
```

Fedora/RHEL:

```
sudo dnf install SDL3-devel
```

Arch Linux:

```
sudo pacman -S sdl3
```

openSUSE:

```
sudo zypper install sdl3-devel
```

These packages install headers to `/usr/include/SDL3/`, libraries to `/usr/lib/` (or `/usr/lib64/`), and CMake config files to the appropriate prefix. The development package (with `-dev` or `-devel`) is required for compiling your own programs.

Building SDL3 from Source

Building from source gives you the latest version and full control over build options. This is recommended if your distribution's package is outdated or if you need specific build features.

Prerequisites:

- CMake 3.5 or later
- A C compiler (GCC, Clang, or MSVC)
- Git (to clone the repository)

Steps:

```
# Clone the repository
git clone https://github.com/libsdl-org/SDL.git
cd SDL

# Optionally, check out a specific release tag
git checkout release-3.2.0

# Configure with CMake
cmake -B build \
    -DCMAKE_BUILD_TYPE=Release \
    -DSDL_TEST=OFF \
    -DSDL_STATIC=OFF

# Build
cmake --build build --parallel

# Install (optional, requires appropriate permissions)
sudo cmake --install build
```

Key CMake options:

Option	Default	Description
SDL_SHARED	ON	Build shared library (.so/.dll/.dylib)
SDL_STATIC	ON	Build static library (.a/.lib)
SDL_TEST	ON	Build test programs and examples
SDL_INSTALL	ON	Enable install target
SDL_LIBC	ON	Include SDL's own libc wrappers (needed for sanitizers)
SDL_PIPEWIRE	AUTO	Enable PipeWire support (Linux audio/video)
SDL_PULSEAUDIO	AUTO	Enable PulseAudio support (Linux audio)
SDL_VULKAN	ON	Enable Vulkan support
SDL_OPENGL	ON	Enable OpenGL support
SDL_OPENGLES	ON	Enable OpenGL ES support
SDL_METAL	AUTO	Enable Metal support (macOS/iOS)
SDL_DIRECT3D	AUTO	Enable Direct3D support (Windows)

For a minimal build with only the features you need, disable unused subsystems. This reduces binary size and dependency count.

Your First SDL3 Program

Let us write a complete, compilable SDL3 program that creates a window and runs an event loop. This is the foundation for everything that follows.

main.cpp:

```
#include <SDL3/SDL.h>
#include <SDL3/SDL_main.h>
#include <iostream>

int main(int argc, char *argv[])
{
    // Initialize the video subsystem
    if (!SDL_Init(SDL_INIT_VIDEO)) {
        SDL_Log("Failed to initialize SDL: %s", SDL_GetError());
        return 1;
    }

    // Create a window (800x600, resizable)
```

```
    SDL_Window *window = SDL_CreateWindow(
        "My First SDL3 Program",
        800, 600,
        SDL_WINDOW_RESIZABLE
    );

    if (!window) {
        SDL_Log("Failed to create window: %s", SDL_GetError());
        SDL_Quit();
        return 1;
    }

    // Main event loop
    bool running = true;
    SDL_Event event;

    while (running) {
        // Process all pending events
        while (SDL_PollEvent(&event)) {
            switch (event.type) {
                case SDL_EVENT_QUIT:
                    running = false;
                    break;
                case SDL_EVENT_KEY_DOWN:
                    if (event.key.key == SDLK_ESCAPE) {
                        running = false;
                    }
                    break;
                default:
                    break;
            }
        }

        // Small delay to prevent 100% CPU usage
        // In a real application, you would render here instead
    }

    // Cleanup
    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.15)
project(FirstSDL3App LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Find SDL3
find_package(SDL3 REQUIRED CONFIG)

# Create the executable
add_executable(FirstSDL3App main.cpp)

# Link against SDL3
target_link_libraries(FirstSDL3App PRIVATE SDL3::SDL3)

# On Windows, copy the SDL3 DLL next to the executable
if(WIN32)
    add_custom_command(TARGET FirstSDL3App POST_BUILD
        COMMAND ${CMAKE_COMMAND} -E copy_if_different
        $<TARGET_FILE:SDL3::SDL3>
        $<TARGET_FILE_DIR:FirstSDL3App>)
endif()
```

Build and run:

```
mkdir build && cd build
cmake ..
cmake --build .
./FirstSDL3App
```

This program demonstrates three fundamental SDL3 concepts:

1. **Initialization**: `SDL_Init(SDL_INIT_VIDEO)` starts the video subsystem. The video subsystem automatically initializes the event subsystem as well, so you do not need to pass `SDL_INIT_EVENTS` separately [2,3].
2. **Window creation**: `SDL_CreateWindow()` creates a native window managed by SDL. The function returns an `SDL_Window*` pointer, or NULL on failure. Always check for errors [4].
3. **Event loop**: `SDL_PollEvent()` retrieves events from the queue one at a time [7]. When it returns false, there are no more events to process. This is the standard pattern for all SDL3 applications.

Notice that `SDL_Init` returns `bool` (true on success, false on failure). This is a key difference from SDL2, where most functions returned 0 on success and

a negative error code on failure [38]. In SDL3, you call `SDL_GetError()` after any function that reports failure to get a human-readable description of what went wrong [30]. Error strings are stored in thread-local storage, so errors in one thread never interfere with another [30].

Error Handling and Logging

SDL3 provides two complementary mechanisms for dealing with errors: the per-thread error string and the structured logging system.

The Error String

Every SDL3 thread maintains its own error string. When an SDL function fails, it sets this string to a descriptive message. You retrieve it with `SDL_GetError()`:

```
if (!window) {
    const char *error = SDL_GetError();
    SDL_Log("Window creation failed: %s", error);
}
```

Important rules:

- The error string is set only on failure. Successful calls do not clear it. Always check the return value before reading the error string.
- The returned pointer is valid until the current thread's error string changes (i.e., until another SDL function fails or you call `SDL_SetError()`).
- Error strings are per-thread, so errors in one thread never interfere with another.

Setting Your Own Errors

When building libraries on top of SDL, you can set your own error messages using `SDL_SetError()`:

```
bool LoadTexture(SDL_Renderer *renderer, const char *path)
{
    SDL_Surface *surface = SDL_LoadBMP(path);
    if (!surface) {
        // SDL_GetError() already contains the loading error
        return false;
    }

    SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, surface);
    SDL_DestroySurface(surface);

    if (!texture) {
        // The error from SDL_CreateTextureFromSurface is still available
        return false;
    }

    return true;
}
```

The Logging System

SDL3's logging system provides categorized, priority-filtered output. It is more powerful than `printf` or `std::cout` for production applications because you can control which messages appear at runtime.

Log priorities (from lowest to highest severity):

Priority	Value	Use Case
`SDL_LOG_PRIORITY_VERBOSE`	1	Detailed trace-level output
`SDL_LOG_PRIORITY_DEBUG`	2	Debugging information
`SDL_LOG_PRIORITY_INFO`	3	General informational messages (default for app/GPU)
`SDL_LOG_PRIORITY_WARN`	4	Warning conditions (default for assert)
`SDL_LOG_PRIORITY_ERROR`	5	Error conditions (default for most categories)
`SDL_LOG_PRIORITY_CRITICAL`	6	Fatal errors

Log categories include `SDL_LOG_CATEGORY_APPLICATION`, `SDL_LOG_CATE-`

GORY_AUDIO, SDL_LOG_CATEGORY_VIDEO, SDL_LOG_CATEGORY_RENDER, SDL_LOG_CATEGORY_INPUT, SDL_LOG_CATEGORY_GPU, and others.

The simplest logging function is `SDL_Log()`, which logs at the INFO level in the APPLICATION category:

```
SDL_Log("Application started successfully");
SDL_Log("Window created: %dx%d", width, height);
```

For other priorities and categories, use `SDL_LogMessage()` or the convenience macros:

```
SDL_LogVerbose(SDL_LOG_CATEGORY_RENDER, "Frame buffer resized to %d x %d", w,
↪  h);
SDL_LogDebug(SDL_LOG_CATEGORY_INPUT, "Key pressed: scancode=%d",
↪  event.key.scancode);
SDL_LogWarn(SDL_LOG_CATEGORY_AUDIO, "Audio device underrun detected");
SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "Failed to load asset: %s", path);
SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION, "Out of memory allocating
↪  texture");
```

Configuring logging at runtime

You can control which messages appear using the `SDL_HINT_LOGGING` hint or by calling `SDL_SetLogPriority()`:

```
// Enable debug-level output for all categories via environment variable
// Set SDL_LOGGING="app=debug,render=verbose,audio=info" before running

// Or configure programmatically (must be done early, before logging)
SDL_SetLogPriority(SDL_LOG_CATEGORY_RENDER, SDL_LOG_PRIORITY_DEBUG);
SDL_SetLogPriority(SDL_LOG_CATEGORY_AUDIO, SDL_LOG_PRIORITY_VERBOSE);
```

CMake Project Setup

A well-structured CMake project is essential for any non-trivial SDL3 application. Let us build a more complete example that demonstrates best practices.

Project structure:

```
MyGame/
├── CMakeLists.txt
├── src/
│   ├── main.cpp
│   ├── Game.h
│   └── Game.cpp
└── assets/
    └── (images, fonts, sounds)
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.15)
project(MyGame VERSION 0.1.0 LANGUAGES CXX)

# Require C++17
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

# Option to use vendored SDL3 (git submodule) vs system package
option(MYGAME_VENDORED "Use vendored SDL3" OFF)

if(MYGAME_VENDORED)
    # Assumes SDL source is in a git submodule at deps/SDL
    add_subdirectory(deps/SDL EXCLUDE_FROM_ALL)
else()
    find_package(SDL3 REQUIRED CONFIG)
endif()

# Source files
file(GLOB_RECURSE GAME_SOURCES "src/*.cpp" "src/*.h")

# Create the executable
add_executable(MyGame ${GAME_SOURCES})

# Link SDL3
target_link_libraries(MyGame PRIVATE SDL3::SDL3)

# Include directories (SDL3 headers are handled by the SDL3::SDL3 target)
target_include_directories(MyGame PRIVATE "${CMAKE_SOURCE_DIR}/src")

# Compiler warnings
if(MSVC)
    target_compile_options(MyGame PRIVATE /W4 /WX)
else()
    target_compile_options(MyGame PRIVATE -Wall -Wextra -Wpedantic -Werror)
endif()

# Output directories
```

```
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin")

# Copy SDL3 DLL on Windows
if(WIN32 AND NOT MYGAME_VENDORED)
    add_custom_command(TARGET MyGame POST_BUILD
        COMMAND ${CMAKE_COMMAND} -E copy_if_different
        $<TARGET_FILE:SDL3::SDL3>
        $<TARGET_FILE_DIR:MyGame>)
endif()

# Install target
install(TARGETS MyGame DESTINATION bin)
```

src/Game.h:

```
#pragma once

#include <SDL3/SDL.h>

class Game {
public:
    Game();
    ~Game();

    // Do not copy or move
    Game(const Game&) = delete;
    Game& operator=(const Game&) = delete;

    bool Initialize();
    void Run();
    void Shutdown();

private:
    bool HandleEvents();
    void Update(float deltaTime);
    void Render();

    SDL_Window* m_window;
    SDL_Renderer* m_renderer;
    bool m_running;
};
```

src/Game.cpp:

```cpp
#include "Game.h"
#include <iostream>

Game::Game() : m_window(nullptr), m_renderer(nullptr), m_running(false) {}

Game::~Game() { Shutdown(); }

bool Game::Initialize()
{
    // Initialize SDL video subsystem
    if (!SDL_Init(SDL_INIT_VIDEO)) {
        SDL_Log("SDL_Init failed: %s", SDL_GetError());
        return false;
    }

    // Create window and renderer together to avoid flicker
    if (!SDL_CreateWindowAndRenderer(
            "MyGame v0.1", 800, 600,
            SDL_WINDOW_RESIZABLE,
            &m_window, &m_renderer)) {
        SDL_Log("Failed to create window/renderer: %s", SDL_GetError());
        return false;
    }

    // Set renderer clear color (dark blue)
    SDL_SetRenderDrawColor(m_renderer, 30, 30, 60, 255);

    m_running = true;
    return true;
}

void Game::Run()
{
    const int TARGET_FPS = 60;
    const int FRAME_TIME = 1000 / TARGET_FPS;
    Uint32 frameStart;
    float deltaTime;

    while (m_running) {
        frameStart = SDL_GetTicks();

        if (!HandleEvents()) break;
        Update(deltaTime);
        Render();

        // Frame timing: cap to target FPS
        Uint32 frameTime = SDL_GetTicks() - frameStart;
        deltaTime = frameTime / 1000.0f;
        if (frameTime < FRAME_TIME) {
            SDL_Delay(FRAME_TIME - frameTime);
```

```
        }
    }
}

void Game::Shutdown()
{
    if (m_renderer) {
        SDL_DestroyRenderer(m_renderer);
        m_renderer = nullptr;
    }
    if (m_window) {
        SDL_DestroyWindow(m_window);
        m_window = nullptr;
    }
    SDL_Quit();
}

bool Game::HandleEvents()
{
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        switch (event.type) {
            case SDL_EVENT_QUIT:
                return false;
            case SDL_EVENT_KEY_DOWN:
                if (event.key.key == SDLK_ESCAPE) {
                    return false;
                }
                break;
            default:
                break;
        }
    }
    return true;
}

void Game::Update(float deltaTime)
{
    // Update game logic here
    (void)deltaTime;
}

void Game::Render()
{
    // Clear the screen
    SDL_RenderClear(m_renderer);

    // Present the rendered frame
    SDL_RenderPresent(m_renderer);
}
```

src/main.cpp:

```cpp
#include "Game.h"

int main(int argc, char *argv[])
{
    (void)argc;
    (void)argv;

    Game game;

    if (!game.Initialize()) {
        SDL_Log("Failed to initialize game");
        return 1;
    }

    game.Run();
    // Shutdown is called automatically by the destructor

    return 0;
}
```

This architecture separates initialization, the main loop, and cleanup into distinct methods. It caps frame rate at 60 FPS using `SDL_Delay()`, calculates delta time for frame-rate-independent updates, and uses RAII (the destructor calls `Shutdown()`) to ensure resources are always freed even if an exception occurs or early return is taken.

The key insight here is `SDL_CreateWindowAndRenderer()`. This convenience function creates both the window and a renderer in one call, avoiding the brief flicker that can occur when you create a window first and then attach a renderer (because SDL may need to recreate the window internally when choosing a rendering backend).

With this foundation in place, you have everything needed to start building real applications. The next chapter covers windows, events, and input handling in depth.

PixelQuest Integration: Chapter 1

Our mini-project, **PixelQuest**, is a simple 2D platformer. In this chapter, we establish the project skeleton that every subsequent chapter will build upon.

The code below is the Game class from the CMake Project Setup section, slightly adapted for PixelQuest:

```
// src/Game.h (PixelQuest v0.1)
#pragma once

#include <SDL3/SDL.h>
#include <SDL3/SDL_main.h>

class Game {
public:
    Game();
    ~Game();

    Game(const Game&) = delete;
    Game& operator=(const Game&) = delete;

    bool Initialize();
    void Run();

private:
    void Shutdown();
    bool HandleEvents();
    void Update(float deltaTime);
    void Render();

    SDL_Window* m_window = nullptr;
    SDL_Renderer* m_renderer = nullptr;
    bool m_running = false;

    // PixelQuest state (expanded in later chapters)
    int m_screenWidth = 800;
    int m_screenHeight = 600;
    float m_lastDelta = 0.0f;
};
```

```cpp
// src/Game.cpp (PixelQuest v0.1)
#include "Game.h"

Game::Game() {}
Game::~Game() { Shutdown(); }

bool Game::Initialize()
{
    SDL_SetAppMetadata("PixelQuest", "0.1.0", "com.example.pixelquest");

    if (!SDL_Init(SDL_INIT_VIDEO)) {
        SDL_Log("SDL_Init failed: %s", SDL_GetError());
        return false;
    }

    if (!SDL_CreateWindowAndRenderer(
            "PixelQuest v0.1", m_screenWidth, m_screenHeight,
            SDL_WINDOW_RESIZABLE,
            &m_window, &m_renderer)) {
        SDL_Log("Failed to create window/renderer: %s", SDL_GetError());
        return false;
    }

    SDL_SetRenderDrawColor(m_renderer, 30, 30, 60, 255);
    m_running = true;
    return true;
}

void Game::Run()
{
    const int TARGET_FPS = 60;
    const int FRAME_TIME = 1000 / TARGET_FPS;
    Uint32 frameStart;
    float deltaTime = 0.0f;

    while (m_running) {
        frameStart = SDL_GetTicks();

        if (!HandleEvents()) break;
        Update(deltaTime);
        Render();

        Uint32 frameTime = SDL_GetTicks() - frameStart;
        deltaTime = frameTime / 1000.0f;
        m_lastDelta = deltaTime;
        if (frameTime < FRAME_TIME) {
            SDL_Delay(FRAME_TIME - frameTime);
        }
    }
}
```

```
51
52  void Game::Shutdown()
53  {
54      if (m_renderer) { SDL_DestroyRenderer(m_renderer); m_renderer = nullptr; }
55      if (m_window)   { SDL_DestroyWindow(m_window);     m_window = nullptr; }
56      SDL_Quit();
57  }
58
59  bool Game::HandleEvents()
60  {
61      SDL_Event event;
62      while (SDL_PollEvent(&event)) {
63          switch (event.type) {
64              case SDL_EVENT_QUIT:
65                  return false;
66              case SDL_EVENT_KEY_DOWN:
67                  if (event.key.key == SDLK_ESCAPE) return false;
68                  break;
69              default: break;
70          }
71      }
72      return true;
73  }
74
75  void Game::Update(float deltaTime) { (void)deltaTime; }
76
77  void Game::Render()
78  {
79      SDL_RenderClear(m_renderer);
80      SDL_RenderPresent(m_renderer);
81  }
```

What we have so far: A window that opens, responds to Escape and the close button, and renders a dark blue background at 60 FPS. In Chapter 2, we will add keyboard-controlled movement and mouse interaction. The Game class structure (Initialize / Run / Shutdown with HandleEvents / Update / Render) remains constant throughout the book; each chapter adds new member variables and logic to these methods.

What we have so far: A window that opens, responds to Escape and the close button, and renders a dark blue background at 60 FPS. In Chapter 2, we will add keyboard-controlled movement and mouse interaction. The Game class structure (Initialize / Run / Shutdown with HandleEvents / Update / Render) remains constant throughout the book; each chapter adds new member variables and logic to these methods.

Chapter 2: Windows, Events, and the Main Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Creating Windows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Window Flags and Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

The Event System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Keyboard Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Mouse Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Text Input and Clipboard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Application Structure: Main Loop vs Callbacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

PixelQuest Integration: Chapter 3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 3: 2D Rendering with SDL_Renderer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

The Renderer Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Drawing Primitives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Working with Textures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Rendering to Textures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Image Loading with SDL_image

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Frame Rate and VSync

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Performance Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 4: Surfaces and Pixel Manipulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Understanding Surfaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Pixel Formats

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Blitting Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Color Keying and Alpha

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Surface Creation and Conversion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

From Surface to Texture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 5: Audio in SDL3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

The New Audio Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Playing Audio with Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Opening and Managing Devices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Audio Mixing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Recording Audio

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

PixelQuest Integration: Chapter 5

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 6: Fonts and Text Rendering with SDL_ttf

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Loading Fonts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Rendering Text to Surfaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Measuring and Fitting Text

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Creating Text Textures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Dynamic Text and Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

PixelQuest Integration: Chapter 6

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Unicode and Internationalization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 7: Input Devices, Gamepads, Joysticks, and Haptics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

The Gamepad API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Opening and Managing Gamepads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Raw Joystick Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Rumble and Haptic Feedback

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Sensors: Accelerometer and Gyroscope

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Input Mapping and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

PixelQuest Integration: Chapter 7

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Building an Input Manager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 8: Timing, Threading, and Synchronization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

High-Resolution Timers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Frame Timing and Delta Time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Creating and Managing Threads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Mutexes and Critical Sections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Semaphores and Condition Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Atomics and Lock-Free Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

PixelQuest Integration: Chapter 8

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Thread-Safe Game Loop Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 9: File I/O, Filesystem, and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

The IOStream System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Memory Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Filesystem Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Configuration Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

PixelQuest Integration: Chapter 9

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Asset Loading Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 10: Graphics Context Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

OpenGL Context Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Vulkan Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Metal Layer Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Swapchain and Presentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

The SDL3 GPU API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

PixelQuest Integration: Chapter 10

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Choosing Your Rendering Path

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 11: Advanced Topics and Platform Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

The Properties System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Hints and Runtime Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Dialog and Filesystem APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Camera Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Clipboard and Drag-and-Drop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Fullscreen, HiDPI, and Multi-Monitor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Platform-Specific Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Chapter 12: Migration from SDL2 to SDL3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Major Breaking Changes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

API Renames and Reorganizations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Audio Migration: Callbacks to Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Rendering Changes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Gamepad and Joystick Changes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Automated Migration Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Common Pitfalls and Gotchas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Conclusion: Building Real Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

A Complete Game Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Debugging SDL3 Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Performance Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Best Practices Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Resources and Community

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Official SDL Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

News and Release Announcements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Historical and Reference Sources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Third-Party Tutorials and Learning Resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Codebase Analysis and Deep Dives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.

Community Discussions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/sdl3thecompletereference.