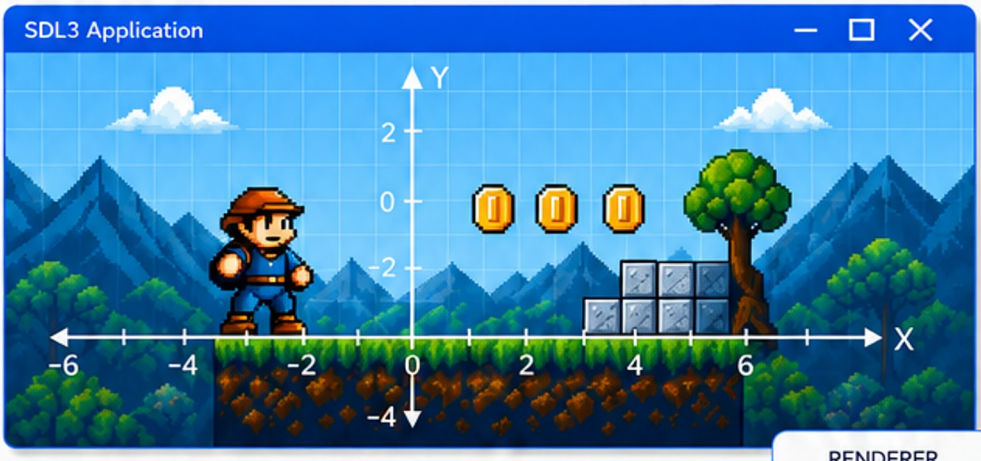
MASTERING SDL3 USING C23

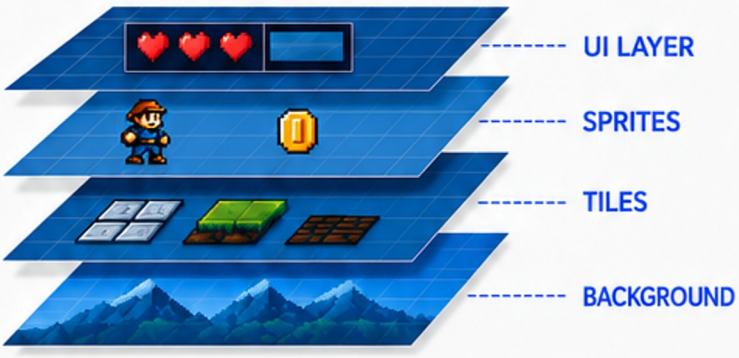
A Practical Visual Guide to Graphics Programming with SDL3, C23, GCC 16.1, and CLion

Prepared by **Ayman Alheraki**

C23 + GCC 16.1 with CLion

```c
#include <SDL3/SDL.h>
#include <stdio.h>

int main(void) {
  SDL_Init(SDL_INIT_VIDEO |
     SDL_INIT_AUDIO | SDL_INIT_EVENTS);

  SDL_Window* win = SDL_CreateWindow(
    "SDL3 Adventure", 960, 540,
     SDL_WINDOW_RESIZABLE);

  SDL_Renderer* ren =
    SDL_CreateRenderer(win, NULL, 0);

  bool running = true;

  while (running) {
     while (SDL_i)
      while (SDL_PollEvent(&ev) {
        if (ev.type == SDL_EVENT_QUIT)
          running = false;
     }

    SDL_SetRenderDrawColor(ren, 30, 36, 48, 255);
    SDL_RenderClear(ren);
    // draw your scene
    SDL_RenderPresent(ren);
  }

  SDL_DestroyRenderer(ren);
  SDL_DestroyWindow(win);
  SDL_Quit();
  return 0;
}
```





RENDERER

SPRITES

TILES

BACKGROUND

For **computer graphics** lovers and **high-efficiency** enthusiasts

RENDERING
High-Performance 2D Rendering

SPRITES
Efficient Sprite Management

INPUT
Keyboard, Mouse, Touch, and Game Controllers

AUDIO
Low-Latency Playback and Mixing

CROSS-PLATFORM
Windows, macOS, Linux, and Beyond with One Codebase

BUILD FAST. RENDER BEAUTIFULLY. SHIP EVERYWHERE.

2026 VISUAL REFERENCE EDITION

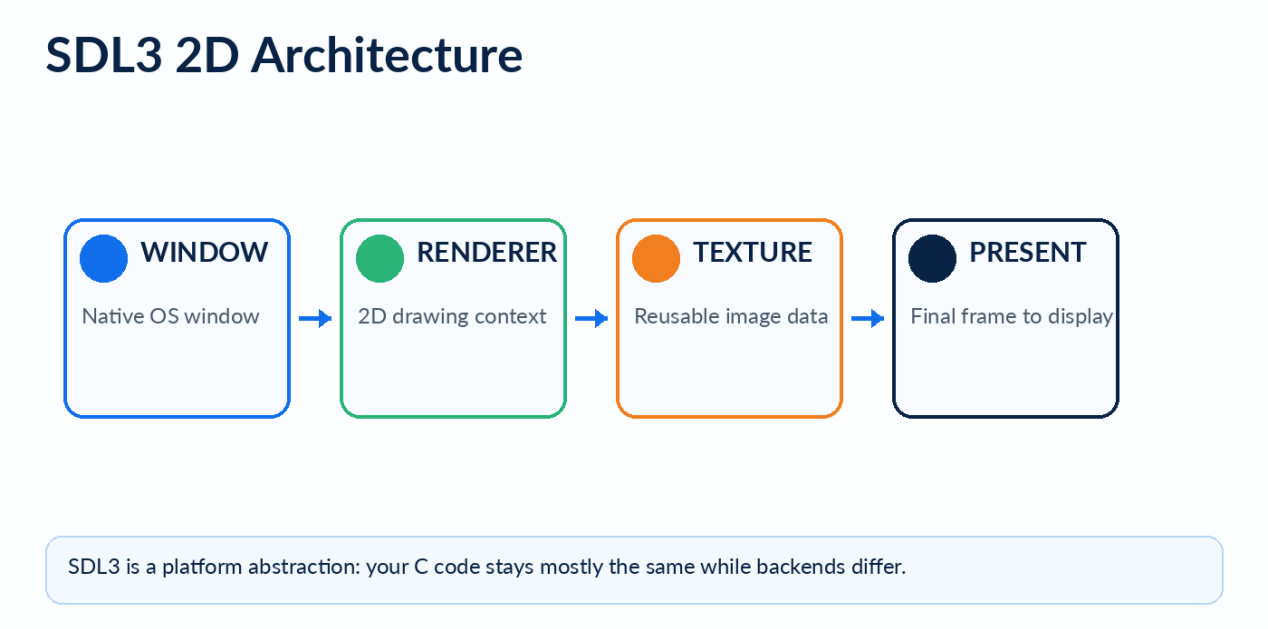
Mastering SDL3 using C23

A Practical Visual Guide to Windows, Rendering, Graphics, Input, Audio, and Cross-Platform Development

Prepared by Ayman Alheraki

Technical baseline: C23, GCC 16.1, CLion 2026.2, CMake, and SDL 3.4.x-oriented APIs.

Audience Computer graphics lovers, C programmers, performance-minded developers, and readers who prefer direct visual learning over unnecessary complexity.



The core objects that appear throughout this book.

Edition note: SDL evolves. The code in this edition intentionally follows SDL3 naming and behavior documented in the SDL 3.2+ / 3.4-era API. When an API changes in a later release, use the official SDL Wiki as the source of truth.

ABOUT THIS EDITION

Practical First. Visual First. C First.

This book is an independent educational guide. SDL is a project of its respective contributors. CLion is a JetBrains product. GCC is part of the GNU toolchain. Product and project names are used for identification and education.

Method

- Each topic starts with a mental model before API details.
- Examples are intentionally small enough to type, understand, and modify.
- The renderer examples use current SDL3-style bool results, SDL_EVENT_* names, float renderer coordinates, modern logical presentation, and SDL_AudioStream-oriented audio.
- Decorative graphics illustrate concepts; the code and official documentation remain authoritative.

Accuracy Baseline

The technical review for this edition uses the official SDL Wiki, SDL CMake documentation, JetBrains CLion 2026.2 documentation, and Fedora package metadata. Fedora 44 provides SDL3 3.4.12 at the time of preparation.

Important Do not copy SDL2 tutorials mechanically into SDL3. Many names, return types, event constants, audio patterns, and helper assumptions changed.

PREFACE

Why This Book Exists

This book was born from a personal need. I wanted a practical guide that explains SDL3 clearly, visually, and step by step while staying close to the C language. I searched for material that was direct enough to build with immediately, yet deep enough to explain why each object and function exists. I did not want the reader to drown in framework architecture before seeing a single useful pixel.

Many university graphics resources are valuable, but they often begin with mathematics, graphics APIs, or higher-level environments that are far from the small, disciplined C programs I wanted to study. That approach has its place; this book serves a different need. It asks: what happens if we choose plain C, a strong cross-platform multimedia library, and a visual-first teaching method?

SDL3 deserves far more attention from people who love computer graphics. It can open windows, create renderers, draw primitives, manage textures, process rich input, handle audio, support high-DPI displays, and travel across major operating systems without turning a small graphics experiment into a platform-engineering project.

My goal is therefore simple: give readers a practical path from the first SDL window to confident 2D graphical programming, with modern tooling and an emphasis on clarity, efficiency, and controlled complexity.

Built Around C23 + GCC 16.1 + CLion 2026.2 + CMake + SDL3. The examples are small by design so the reader always knows which line caused which visible result.

READER GUIDE

How to Use This Book

Read in Three Passes

- Pass 1: scan the diagrams and mental models.
- Pass 2: type the code, do not merely read it.
- Pass 3: modify one variable, color, coordinate, or asset at a time and observe the result.

Keep One Playground Project

- Use one repository with small chapter targets or folders.
- Keep assets under an assets/ directory.
- Use Debug while learning and Release when measuring performance.
- Commit after each working milestone.

Recommended Project Layout

PROJECT TREE • COPY-READY

```
SDL3Playground/
├── CMakeLists.txt
├── src/
│   ├── main.c
│   └── demo_*.c
├── assets/
│   ├── images/
│   ├── fonts/
│   └── audio/
└── README.md
```

Rule Never proceed from a failed resource creation. Log SDL_GetError(), clean up what already exists, and return to a known state.

CONTENTS

Book Roadmap — Part I to III

CONTENTS

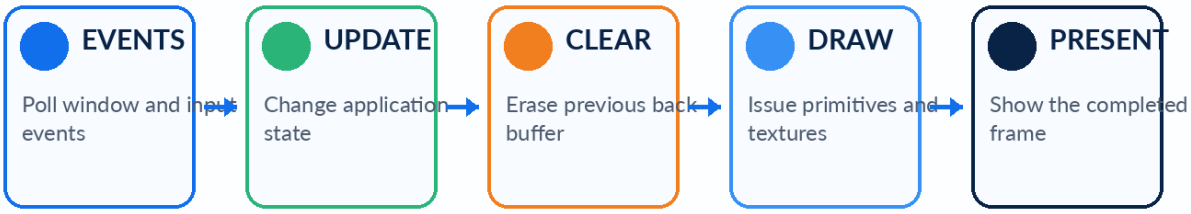
Book Roadmap — Part IV, Project, and Appendices

Appendices

ROADMAP

From Window to Finished Application

One Frame: The Mental Model



The order is simple: input -> state -> render -> present. Repeat until quit.

The recurring frame loop connects every major subject in the book.

Foundation

- Toolchain
- Window + renderer
- Lifecycle
- Frame loop
- Coordinates

Capability

- Images + textures
- Animation + text
- Input + timing
- Audio
- Release + shipping

North Star At every stage you should be able to answer four questions: What owns this resource? When is it updated? When is it drawn? When is it destroyed?

CHAPTER 1 — FOUNDATIONS

Toolchain Setup: GCC 16.1, CLion, CMake, and SDL3

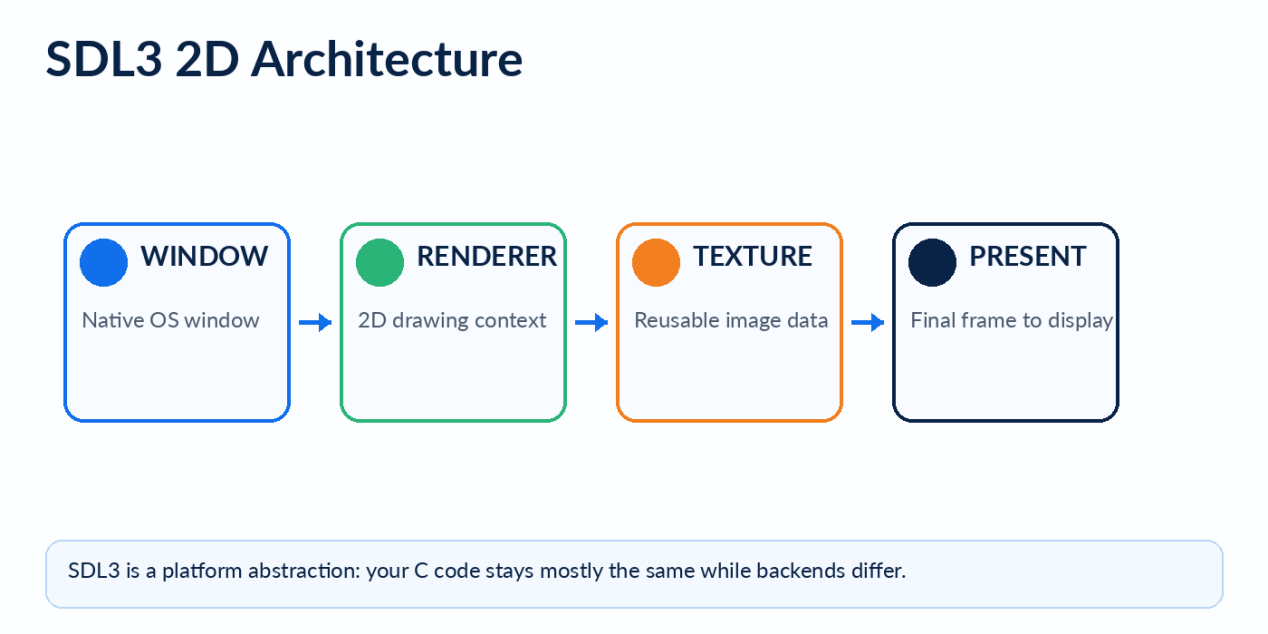
A reliable graphics workflow starts before the first pixel is drawn. This chapter builds a predictable C23 toolchain around GCC 16.1, CLion 2026.2, CMake, Ninja, and SDL3. The goal is not to memorize IDE dialogs; it is to understand which component owns which job so you can diagnose problems instead of guessing.

Core Mental Model

- GCC compiles C source and links the final executable.
- CMake describes targets, dependencies, compile options, and link relationships.
- CLion is the editor/debugger front end; it should point at your chosen system toolchain.
- SDL3 development packages supply headers, libraries, CMake package metadata, and usually pkg-config metadata.

Key APIs / Tools

- find_package(SDL3 CONFIG REQUIRED)
- SDL3::SDL3
- CMAKE_C_STANDARD 23
- CMAKE_C_COMPILER
- CMake Profiles in CLion



Visual model for Chapter 1.

CHAPTER 1 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`find_package(SDL3 CONFIG REQUIRED)` — is a central function, type, target, or configuration point for this chapter.

`SDL3::SDL3` — is a central function, type, target, or configuration point for this chapter.

`CMAKE_C_STANDARD 23` — is a central function, type, target, or configuration point for this chapter.

`CMAKE_C_COMPILER` — is a central function, type, target, or configuration point for this chapter.

`CMake Profiles in CLion` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 1 — WORKING CODE

A Minimal Pattern You Can Build On

Example

CMAKE • COPY-READY

```cmake
cmake_minimum_required(VERSION 3.25)
project(SDL3BookDemo LANGUAGES C)

set(CMAKE_C_STANDARD 23)
set(CMAKE_C_STANDARD_REQUIRED ON)
set(CMAKE_C_EXTENSIONS OFF)

find_package(SDL3 CONFIG REQUIRED)
add_executable(SDL3BookDemo src/main.c)
target_link_libraries(SDL3BookDemo PRIVATE SDL3::SDL3)

target_compile_options(SDL3BookDemo PRIVATE
    $<$<C_COMPILER_ID:GNU>:-Wall -Wextra -Wpedantic>)
```

What to Notice

- On Fedora 44 install SDL3-devel.
- Verify gcc, cmake, ninja, and pkg-config before opening the IDE.
- Set /usr/bin/gcc as the C compiler in the CLion toolchain.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 1 — WORKFLOW

The Reliable Step-by-Step Workflow

1. On Fedora 44 install SDL3-devel.
2. Verify gcc, cmake, ninja, and pkg-config before opening the IDE.
3. Set /usr/bin/gcc as the C compiler in the CLion toolchain.
4. Use one Debug and one Release CMake profile.
5. Let imported CMake targets propagate include directories and link details.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 1 — WORKING CODE

A Minimal Pattern You Can Build On

Example

TERMINAL • COPY-READY

```
gcc --version
cmake --version
ninja --version
pkg-config --modversion sdl3
```

```
# Fedora 44
sudo dnf install SDL3-devel
```

What to Notice

- On Fedora 44 install SDL3-devel.
- Verify gcc, cmake, ninja, and pkg-config before opening the IDE.
- Set /usr/bin/gcc as the C compiler in the CLion toolchain.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 1 — FAILURE MODES

Mistakes That Waste the Most Time

1. Hard-coding /usr/include/SDL3 or library paths makes projects fragile

Hard-coding /usr/include/SDL3 or library paths makes projects fragile.

2. Mixing multiple SDL installations can make CMake find a different library than the executable loads

Mixing multiple SDL installations can make CMake find a different library than the executable loads.

3. A successful compile is not proof that runtime shared libraries are discoverable

A successful compile is not proof that runtime shared libraries are discoverable.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 1 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain gCC compiles C source and links the final executable..
- I can explain cMake describes targets, dependencies, compile options, and link relationships..
- I can explain cLion is the editor/debugger front end; it should point at your chosen system toolchain..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Create an empty C project, configure both Debug and Release profiles, print the SDL runtime version, and verify the same source builds from CLion and from a terminal.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 2 — FOUNDATIONS

Your First SDL3 Window

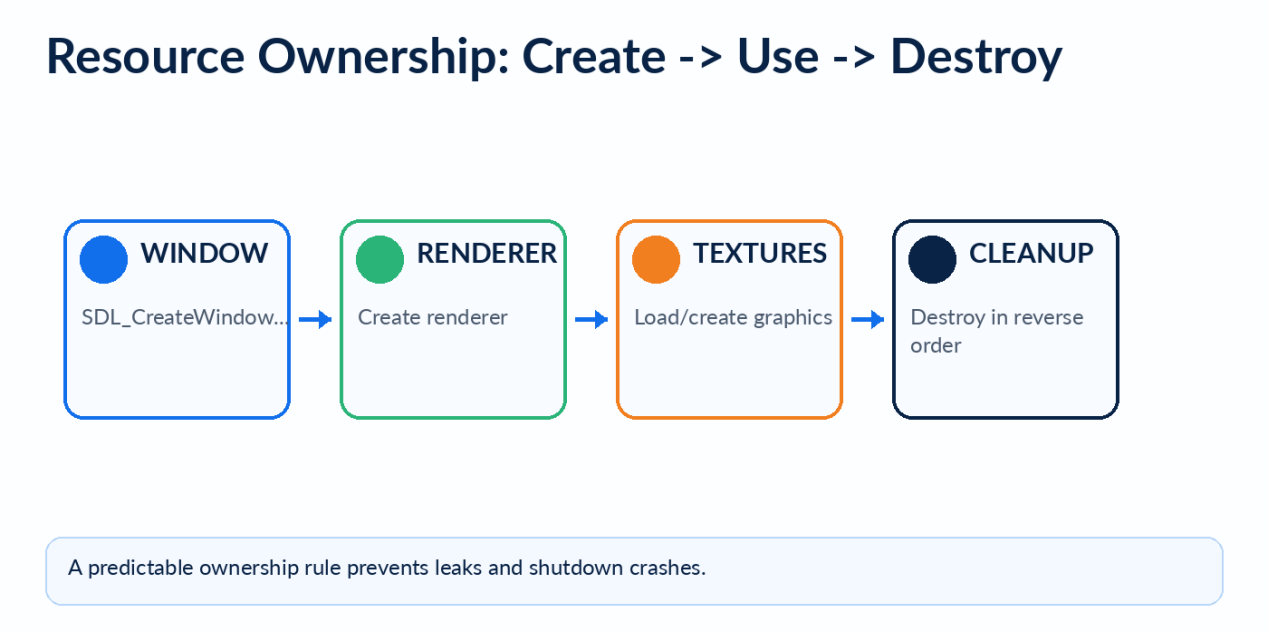
The first SDL3 program should teach the complete lifetime of a graphical application: initialize SDL, create a window and renderer, process events, draw a frame, then release resources in a clear reverse order. This tiny pattern is the skeleton behind everything later in the book.

Core Mental Model

- SDL_Init returns bool in SDL3: true means success.
- SDL_CreateWindowAndRenderer is a convenient one-call setup for common 2D programs.
- The application remains responsive only while it pumps and processes events.
- Cleanup is explicit; resources are not garbage collected.

Key APIs / Tools

- SDL_Init
- SDL_CreateWindow
- SDL_CreateWindowAndRenderer
- SDL_DestroyWindow
- SDL_Quit



Visual model for Chapter 2.

CHAPTER 2 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_Init` — initializes selected SDL subsystems and reports success as bool.

`SDL_CreateWindow` — is a central function, type, target, or configuration point for this chapter.

`SDL_CreateWindowAndRenderer` — is a central function, type, target, or configuration point for this chapter.

`SDL_DestroyWindow` — is a central function, type, target, or configuration point for this chapter.

`SDL_Quit` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 2 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

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

int main(int argc, char **argv)
{
    (void)argc; (void)argv;
    if (!SDL_Init(SDL_INIT_VIDEO)) {
        SDL_Log("SDL_Init failed: %s", SDL_GetError());
        return 1;
    }

    SDL_Window *window = NULL;
    SDL_Renderer *renderer = NULL;
    if (!SDL_CreateWindowAndRenderer(
            "SDL3 using C23", 960, 540,
            SDL_WINDOW_RESIZABLE | SDL_WINDOW_HIGH_PIXEL_DENSITY,
            &window, &renderer)) {
        SDL_Log("CreateWindowAndRenderer: %s", SDL_GetError());
        SDL_Quit();
        return 1;
    }

    bool running = true;
    SDL_Event e;
    while (running) {
        while (SDL_PollEvent(&e)) {
            if (e.type == SDL_EVENT_QUIT) running = false;
            if (e.type == SDL_EVENT_KEY_DOWN && e.key.key == SDLK_ESCAPE)
                running = false;
        }
        SDL_SetRenderDrawColor(renderer, 18, 24, 38, 255);
        SDL_RenderClear(renderer);
        SDL_RenderPresent(renderer);
    }

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

What to Notice

- Initialize only the subsystems you need.
- Create the window on the main thread.
- Keep a run flag and event loop.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 2 — WORKFLOW

The Reliable Step-by-Step Workflow

6. Initialize only the subsystems you need.
7. Create the window on the main thread.
8. Keep a run flag and event loop.
9. Treat the close request and Escape key as ordinary state changes.
10. Destroy renderer before window, then call SDL_Quit.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 2 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_Event e;
while (SDL_PollEvent(&e)) {
    switch (e.type) {
    case SDL_EVENT_QUIT:
        running = false;
        break;
    case SDL_EVENT_KEY_DOWN:
        if (e.key.key == SDLK_ESCAPE) running = false;
        break;
    case SDL_EVENT_WINDOW_RESIZED:
        SDL_Log("window: %dx%d", e.window.data1, e.window.data2);
        break;
    default:
        break;
    }
}
```

What to Notice

- Initialize only the subsystems you need.
- Create the window on the main thread.
- Keep a run flag and event loop.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 2 — FAILURE MODES

Mistakes That Waste the Most Time

1. Testing SDL_Init with < 0 is SDL2-style logic and is wrong for the SDL3 bool return

Testing SDL_Init with < 0 is SDL2-style logic and is wrong for the SDL3 bool return.

2. A window that does not process events will look frozen to the operating system

A window that does not process events will look frozen to the operating system.

3. Do not continue after a failed window or renderer creation

Do not continue after a failed window or renderer creation.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did

the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 2 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain sDL_Init returns bool in SDL3: true means success..
- I can explain sDL_CreateWindowAndRenderer is a convenient one-call setup for common 2D programs..
- I can explain the application remains responsive only while it pumps and processes events..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Open a 960x540 resizable high-pixel-density window. Add Escape-to-quit, print resize events, and change the clear color when the window gains or loses keyboard focus.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 3 — FOUNDATIONS

The Renderer: Your 2D Drawing Context

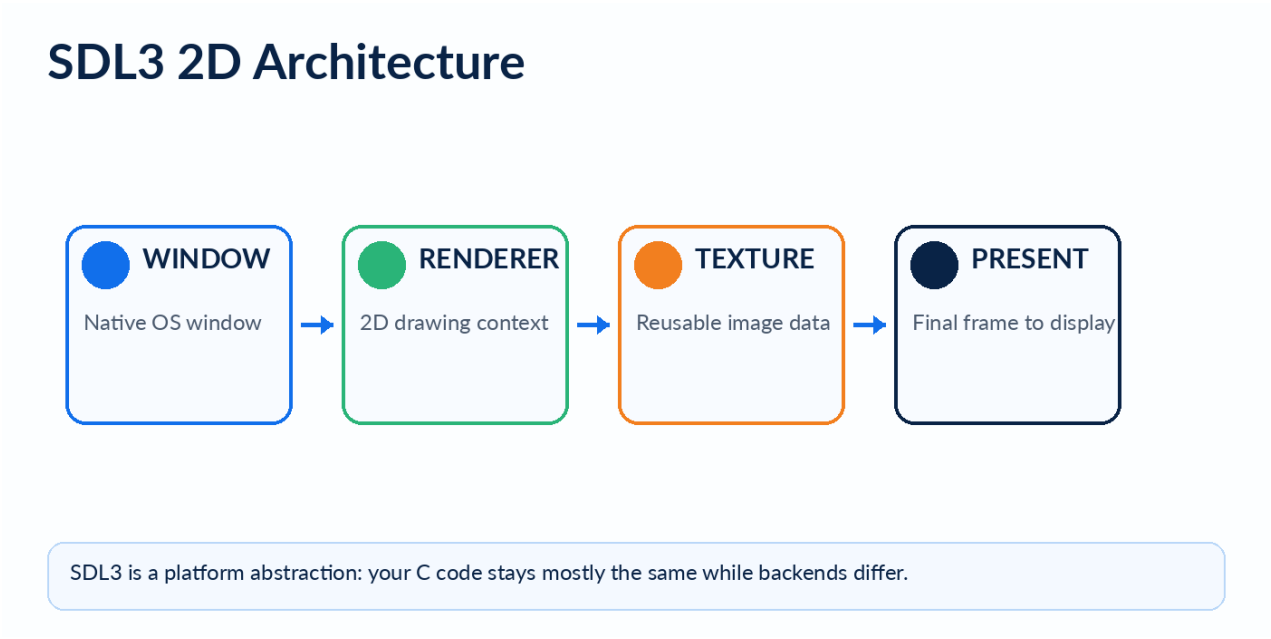
SDL_Renderer is the center of SDL3's high-level 2D API. You submit drawing commands to it, targeting either the window or a texture. The renderer chooses an available backend and manages presentation details while you work in a small, direct API.

Core Mental Model

- The window is an OS surface; the renderer is the drawing context.
- Most 2D rendering functions operate on the main thread.
- The renderer maintains state: current draw color, viewport, clip rectangle, scale, logical presentation, and target.
- Textures belong to the renderer that created them.

Key APIs / Tools

- SDL_CreateRenderer
- SDL_GetRendererName
- SDL_SetRenderDrawColor
- SDL_RenderClear
- SDL_RenderPresent



Visual model for Chapter 3.

CHAPTER 3 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_CreateRenderer` — is a central function, type, target, or configuration point for this chapter.

`SDL_GetRendererName` — is a central function, type, target, or configuration point for this chapter.

`SDL_SetRenderDrawColor` — is a central function, type, target, or configuration point for this chapter.

`SDL_RenderClear` — is a central function, type, target, or configuration point for this chapter.

`SDL_RenderPresent` — presents the current renderer back buffer to the display.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 3 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_SetRenderDrawColor(renderer, 14, 22, 38, 255);
SDL_RenderClear(renderer);

SDL_SetRenderDrawColor(renderer, 80, 190, 255, 255);
SDL_RenderLine(renderer, 40.0f, 40.0f, 420.0f, 190.0f);

SDL_FRect box = { 100.0f, 110.0f, 180.0f, 90.0f };
SDL_SetRenderDrawColor(renderer, 55, 210, 130, 255);
SDL_RenderRect(renderer, &box);

SDL_FRect fill = { 360.0f, 250.0f, 150.0f, 80.0f };
SDL_SetRenderDrawColor(renderer, 245, 125, 45, 255);
SDL_RenderFillRect(renderer, &fill);

SDL_RenderPresent(renderer);
```

What to Notice

- Choose a draw color.
- Clear the current target.
- Draw primitives and textures in back-to-front order.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 3 — WORKFLOW

The Reliable Step-by-Step Workflow

11. Choose a draw color.
12. Clear the current target.
13. Draw primitives and textures in back-to-front order.
14. Present once at the end of the frame.
15. Repeat while the application is running.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 3 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
const char *name = SDL_GetRendererName(renderer);
SDL_Log("Renderer: %s", name ? name : "unknown");

SDL_SetRenderDrawColor(renderer, 10, 18, 30, 255);
SDL_RenderClear(renderer);
// draw...
SDL_RenderPresent(renderer);
```

What to Notice

- Choose a draw color.
- Clear the current target.
- Draw primitives and textures in back-to-front order.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 3 — FAILURE MODES

Mistakes That Waste the Most Time

1. Presenting after every object destroys batching opportunities and complicates animation

Presenting after every object destroys batching opportunities and complicates animation.

2. Using a texture with a different renderer is invalid

Using a texture with a different renderer is invalid.

3. State changes are cheap conceptually but should still be organized

State changes are cheap conceptually but should still be organized.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 3 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain the window is an OS surface; the renderer is the drawing context..
- I can explain most 2D rendering functions operate on the main thread..
- I can explain the renderer maintains state: current draw color, viewport, clip rectangle, scale, logical presentation, and target..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Print the active renderer name, draw three colored regions, and experiment with changing the clear color once per second.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 4 — FOUNDATIONS

The Application Loop: Events, Update, Render

A graphical program becomes easy to reason about when one frame has a predictable shape. SDL does not force a game engine architecture on you; a disciplined event-update-render loop is enough for many tools, demos, visualizations, and 2D games.

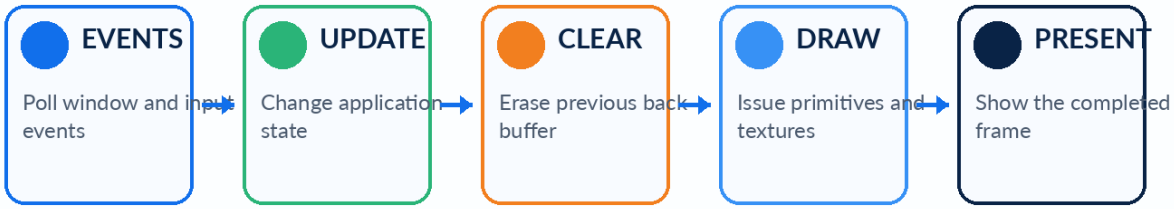
Core Mental Model

- Events describe discrete changes; state describes what is true now.
- Update logic changes your model; render logic observes it.
- One render present per frame keeps output coherent.
- A clean loop makes later timing, animation, and input chapters almost mechanical.

Key APIs / Tools

- SDL_PollEvent
- SDL_PumpEvents
- SDL_RenderPresent
- SDL_Delay
- SDL_GetTicksNS

One Frame: The Mental Model



The order is simple: input -> state -> render -> present. Repeat until quit.

Visual model for Chapter 4.

CHAPTER 4 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_PollEvent` — removes the next pending event and writes it into SDL_Event.

`SDL_PumpEvents` — is a central function, type, target, or configuration point for this chapter.

`SDL_RenderPresent` — presents the current renderer back buffer to the display.

`SDL_Delay` — is a central function, type, target, or configuration point for this chapter.

`SDL_GetTicksNS` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 4 — WORKING CODE

A Minimal Pattern You Can Build On

Example

CODE • COPY-READY

```
while (running) {
    HandleEvents(&running);
    ReadInput();
    Update(dt);
    Render(renderer);
}
```

What to Notice

- Poll all queued events.
- Read continuous input state.
- Update application state using delta time.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 4 — WORKFLOW

The Reliable Step-by-Step Workflow

16. Poll all queued events.
17. Read continuous input state.
18. Update application state using delta time.
19. Clear and render the complete frame.
20. Present exactly once.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 4 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
static Uint64 previous_ns = 0;
Uint64 now_ns = SDL_GetTicksNS();
if (previous_ns == 0) previous_ns = now_ns;

float dt = (float)(now_ns - previous_ns) / 1000000000.0f;
previous_ns = now_ns;
if (dt > 0.05f) dt = 0.05f;   // clamp debugger/pause spikes

player_x += player_speed * dt;
```

What to Notice

- Poll all queued events.
- Read continuous input state.
- Update application state using delta time.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 4 — FAILURE MODES

Mistakes That Waste the Most Time

1. Putting game logic inside every event handler makes continuous motion difficult

Putting game logic inside every event handler makes continuous motion difficult.

2. Calling SDL_Delay(16) does not guarantee exactly 60 FPS

Calling SDL_Delay(16) does not guarantee exactly 60 FPS.

3. Expensive file I/O in the frame loop causes visible stutter

Expensive file I/O in the frame loop causes visible stutter.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 4 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain events describe discrete changes; state describes what is true now..
- I can explain update logic changes your model; render logic observes it..
- I can explain one render present per frame keeps output coherent..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Refactor the first-window program into functions HandleEvents, Update, Render, and Shutdown. Make no rendering call from HandleEvents.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 5 — CORE GRAPHICS

Colors, Coordinates, and the Rendering Flow

Graphics become simple when you stop thinking in abstract “screen positions” and instead use a precise coordinate system. SDL’s 2D renderer uses the top-left as the origin, X increases rightward, and Y increases downward. Colors are usually expressed as RGBA channels.

Core Mental Model

- Renderer coordinates are floating-point for many SDL3 drawing calls.
- RGBA uses red, green, blue, and alpha channels.
- Alpha has an effect only when a compatible blend mode is used.
- Clear uses the current draw color and affects the current render target.

Key APIs / Tools

- SDL_SetRenderDrawColor
- SDL_SetRenderDrawColorFloat
- SDL_RenderPoint
- SDL_RenderLine
- SDL_RenderClear

Visual model for Chapter 5.

CHAPTER 5 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_SetRenderDrawColor` — is a central function, type, target, or configuration point for this chapter.

`SDL_SetRenderDrawColorFloat` — is a central function, type, target, or configuration point for this chapter.

`SDL_RenderPoint` — is a central function, type, target, or configuration point for this chapter.

`SDL_RenderLine` — is a central function, type, target, or configuration point for this chapter.

`SDL_RenderClear` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 5 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_SetRenderDrawColor(renderer, 14, 22, 38, 255);
SDL_RenderClear(renderer);

SDL_SetRenderDrawColor(renderer, 80, 190, 255, 255);
SDL_RenderLine(renderer, 40.0f, 40.0f, 420.0f, 190.0f);

SDL_FRect box = { 100.0f, 110.0f, 180.0f, 90.0f };
SDL_SetRenderDrawColor(renderer, 55, 210, 130, 255);
SDL_RenderRect(renderer, &box);

SDL_FRect fill = { 360.0f, 250.0f, 150.0f, 80.0f };
SDL_SetRenderDrawColor(renderer, 245, 125, 45, 255);
SDL_RenderFillRect(renderer, &fill);

SDL_RenderPresent(renderer);
```

What to Notice

- Choose a background color and clear.
- Choose a foreground color.
- Draw geometry in window or logical coordinates.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 5 — WORKFLOW

The Reliable Step-by-Step Workflow

21. Choose a background color and clear.
22. Choose a foreground color.
23. Draw geometry in window or logical coordinates.
24. Change color only when the next group needs it.
25. Present the completed frame.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 5 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_SetRenderDrawColorFloat(renderer, 0.1f, 0.2f, 0.35f, 1.0f);
SDL_RenderClear(renderer);
SDL_RenderPoint(renderer, 20.5f, 20.5f);
SDL_RenderLine(renderer, 50.0f, 50.0f, 250.0f, 130.0f);
```

What to Notice

- Choose a background color and clear.
- Choose a foreground color.
- Draw geometry in window or logical coordinates.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 5 — FAILURE MODES

Mistakes That Waste the Most Time

1. Y grows downward, which surprises developers coming from mathematical graphs

Y grows downward, which surprises developers coming from mathematical graphs.

2. Drawing before Clear erases the work if Clear is called afterward

Drawing before Clear erases the work if Clear is called afterward.

3. Alpha 0 does not automatically mean invisible unless blending is configured appropriately

Alpha 0 does not automatically mean invisible unless blending is configured appropriately.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 5 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain renderer coordinates are floating-point for many SDL3 drawing calls..
- I can explain rGBA uses red, green, blue, and alpha channels..
- I can explain alpha has an effect only when a compatible blend mode is used..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Draw a crosshair at the center, four corner markers, and a rectangle whose position is entered as percentages of the window size.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 6 — CORE GRAPHICS

Points, Lines, Rectangles, and Primitive Drawing

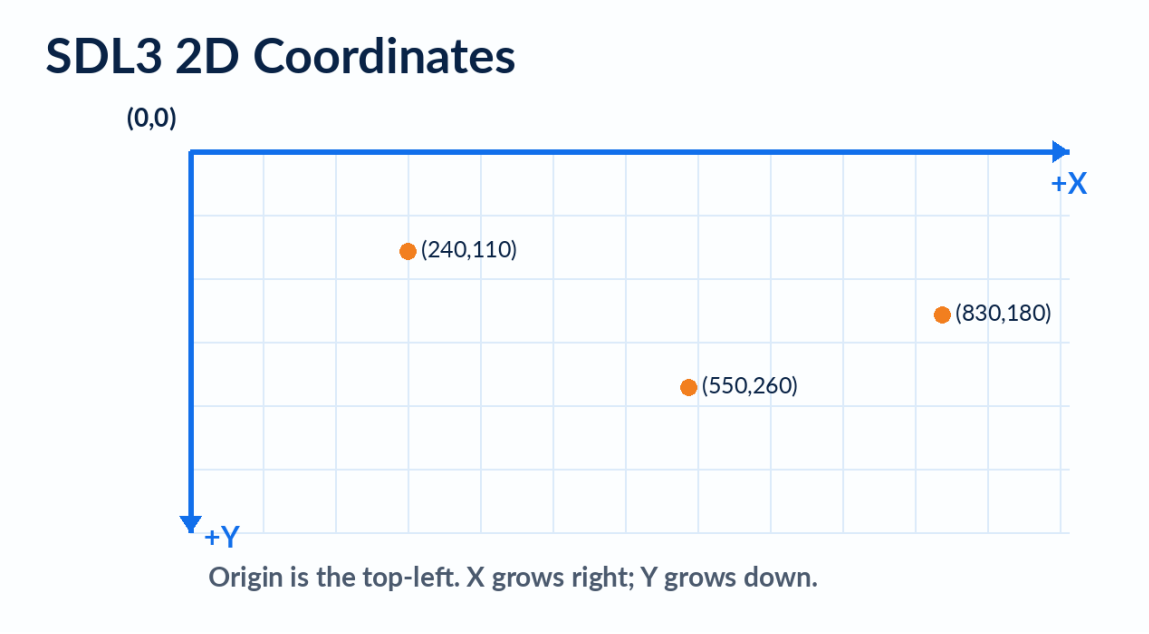
Primitives are more than beginner exercises. They are excellent for debug overlays, graphs, editors, selection boxes, collision visualization, grids, rulers, handles, and instrumentation. Mastering them makes later graphical tools easier to build.

Core Mental Model

- Use SDL_FPoint and SDL_FRect when you want subpixel-friendly geometry.
- Line and rectangle drawing follows the current renderer color.
- Batch related geometry conceptually and minimize unnecessary state changes.
- Complex shapes can often be decomposed into lines and filled rectangles.

Key APIs / Tools

- SDL_RenderPoint
- SDL_RenderPoints
- SDL_RenderLine
- SDL_RenderLines
- SDL_RenderRect
- SDL_RenderFillRect



Visual model for Chapter 6.

CHAPTER 6 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

SDL_RenderPoint — is a central function, type, target, or configuration point for this chapter.

SDL_RenderPoints — is a central function, type, target, or configuration point for this chapter.

SDL_RenderLine — is a central function, type, target, or configuration point for this chapter.

SDL_RenderLines — is a central function, type, target, or configuration point for this chapter.

SDL_RenderRect — is a central function, type, target, or configuration point for this chapter.

SDL_RenderFillRect — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 6 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_SetRenderDrawColor(renderer, 14, 22, 38, 255);
SDL_RenderClear(renderer);
```

```c
SDL_SetRenderDrawColor(renderer, 80, 190, 255, 255);
SDL_RenderLine(renderer, 40.0f, 40.0f, 420.0f, 190.0f);

SDL_FRect box = { 100.0f, 110.0f, 180.0f, 90.0f };
SDL_SetRenderDrawColor(renderer, 55, 210, 130, 255);
SDL_RenderRect(renderer, &box);

SDL_FRect fill = { 360.0f, 250.0f, 150.0f, 80.0f };
SDL_SetRenderDrawColor(renderer, 245, 125, 45, 255);
SDL_RenderFillRect(renderer, &fill);

SDL_RenderPresent(renderer);
```

What to Notice

- Clear the frame.
- Draw background guides first.
- Draw filled regions.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 6 — WORKFLOW

The Reliable Step-by-Step Workflow

26. Clear the frame.
27. Draw background guides first.
28. Draw filled regions.
29. Draw outlines and handles last.
30. Present once.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 6 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_FPoint polyline[] = {
    {20, 180}, {80, 90}, {150, 140}, {240, 60}, {330, 120}
};
SDL_SetRenderDrawColor(renderer, 100, 210, 255, 255);
SDL_RenderLines(renderer, polyline, 5);
```

What to Notice

- Clear the frame.
- Draw background guides first.
- Draw filled regions.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 6 — FAILURE MODES

Mistakes That Waste the Most Time

1. Integer-looking coordinates can still be passed as float values

Integer-looking coordinates can still be passed as float values.

2. Outlines and fills may overlap; draw order decides what remains visible

Outlines and fills may overlap; draw order decides what remains visible.

3. Do not rebuild static geometry calculations every frame if they can be cached

Do not rebuild static geometry calculations every frame if they can be cached.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 6 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain use SDL_FPoint and SDL_FRect when you want subpixel-friendly geometry..
- I can explain line and rectangle drawing follows the current renderer color..
- I can explain batch related geometry conceptually and minimize unnecessary state changes..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Build a tiny vector drawing canvas with a grid, two rectangles, a selection outline, and draggable corner handles.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 7 — CORE GRAPHICS

Surfaces, Textures, and Image Loading

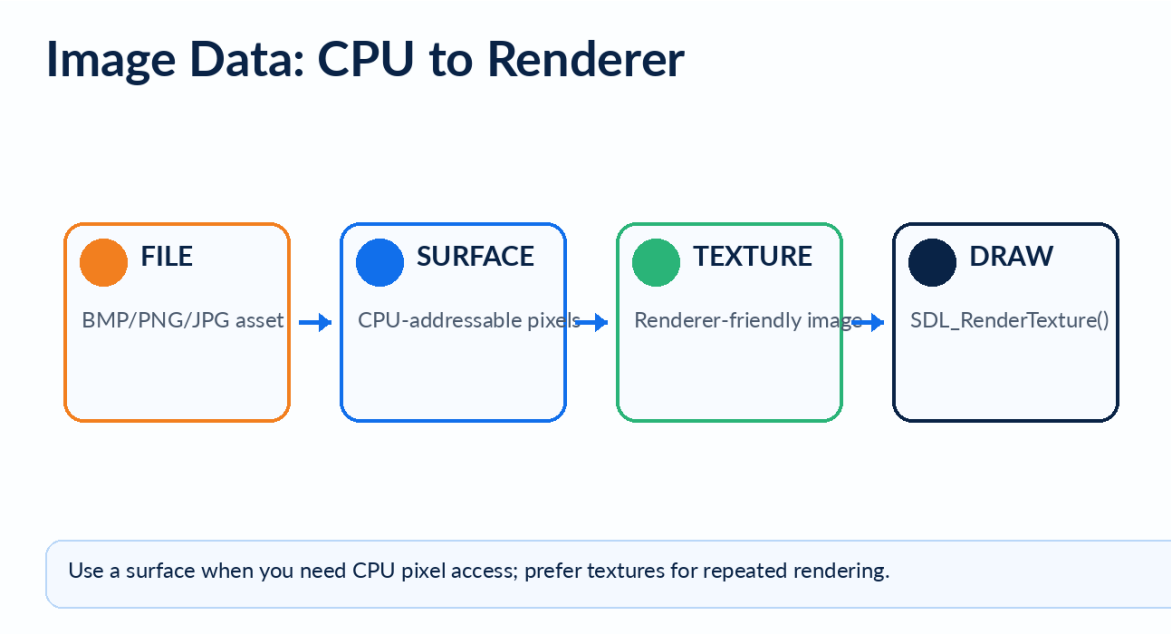
SDL distinguishes CPU-addressable pixel buffers from renderer-friendly textures. An SDL_Surface is ideal when you need to inspect or modify pixels on the CPU. An SDL_Texture is what you normally keep around for repeated drawing.

Core Mental Model

- Surfaces live in ordinary memory and expose pixel data.
- Textures are optimized representations owned by a renderer.
- SDL_CreateTextureFromSurface copies data; the surface may be destroyed afterward.
- SDL_image loads many formats and can create textures directly.

Key APIs / Tools

- SDL_LoadBMP
- SDL_DestroySurface
- SDL_CreateTextureFromSurface
- SDL_RenderTexture
- IMG_LoadTexture



Visual model for Chapter 7.

CHAPTER 7 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_LoadBMP` — is a central function, type, target, or configuration point for this chapter.

`SDL_DestroySurface` — is a central function, type, target, or configuration point for this chapter.

`SDL_CreateTextureFromSurface` — is a central function, type, target, or configuration point for this chapter.

`SDL_RenderTexture` — copies a texture region to the active rendering target.

`IMG_LoadTexture` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 7 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_Surface *surface = SDL_LoadBMP("assets/picture.bmp");
if (!surface) {
    SDL_Log("SDL_LoadBMP: %s", SDL_GetError());
    return false;
}
```

```c
SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, surface);
SDL_DestroySurface(surface);
if (!texture) {
    SDL_Log("SDL_CreateTextureFromSurface: %s", SDL_GetError());
    return false;
}

SDL_FRect dst = { 80.0f, 60.0f, 512.0f, 288.0f };
SDL_RenderTexture(renderer, texture, NULL, &dst);
SDL_DestroyTexture(texture);
```

What to Notice

- Load or create a surface.
- Inspect/modify it only if necessary.
- Create a texture from the surface.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 7 — WORKFLOW

The Reliable Step-by-Step Workflow

31. Load or create a surface.
32. Inspect/modify it only if necessary.
33. Create a texture from the surface.
34. Destroy the surface once the texture owns the needed image data.
35. Reuse the texture across frames and destroy it on shutdown.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 7 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
#include <SDL3_image/SDL_image.h>

SDL_Texture *texture = IMG_LoadTexture(renderer, "assets/art.png");
if (!texture) {
    SDL_Log("IMG_LoadTexture: %s", SDL_GetError());
}
// ... render texture repeatedly ...
SDL_DestroyTexture(texture);
```

What to Notice

- Load or create a surface.
- Inspect/modify it only if necessary.
- Create a texture from the surface.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 7 — FAILURE MODES

Mistakes That Waste the Most Time

1. Do not access surface fields after destroying the surface

Do not access surface fields after destroying the surface.

2. Repeatedly loading an image every frame is a severe performance mistake

Repeatedly loading an image every frame is a severe performance mistake.

3. Relative asset paths depend on the process working directory unless you build a stronger asset locator

Relative asset paths depend on the process working directory unless you build a stronger asset locator.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 7 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain surfaces live in ordinary memory and expose pixel data..
- I can explain textures are optimized representations owned by a renderer..
- I can explain sDL_CreateTextureFromSurface copies data; the surface may be destroyed afterward..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Load one BMP with SDL_LoadBMP and one PNG with SDL_image. Display both, resize them independently, and add an error fallback rectangle if either asset fails.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 7 — DEEPER DESIGN

Going One Layer Deeper Without Losing Simplicity

For large applications, separate asset decoding from render-time use. Build a small texture cache keyed by asset path. The cache should own SDL_Texture pointers, return existing textures for repeated requests, and destroy everything before the renderer. This simple resource layer eliminates duplicate loads and makes scene code cleaner.

Design Boundary

Create small helpers when they remove repeated ownership or repeated error checks. Avoid wrappers that rename every SDL function without adding a real policy. The official SDL documentation should remain easy to map onto your codebase.

Measure, Do Not Guess

- Time real workloads, not empty loops.
- Profile asset loading separately from frame rendering.
- Test on the weakest target you intend to support.
- Keep debug overlays available in development builds.

Good Abstraction An abstraction is earning its place when it makes ownership, lifetime, or intent clearer — not merely when it reduces the number of SDL names visible in your code.

CHAPTER 8 — CORE GRAPHICS

Sprites, Sprite Sheets, and Animation

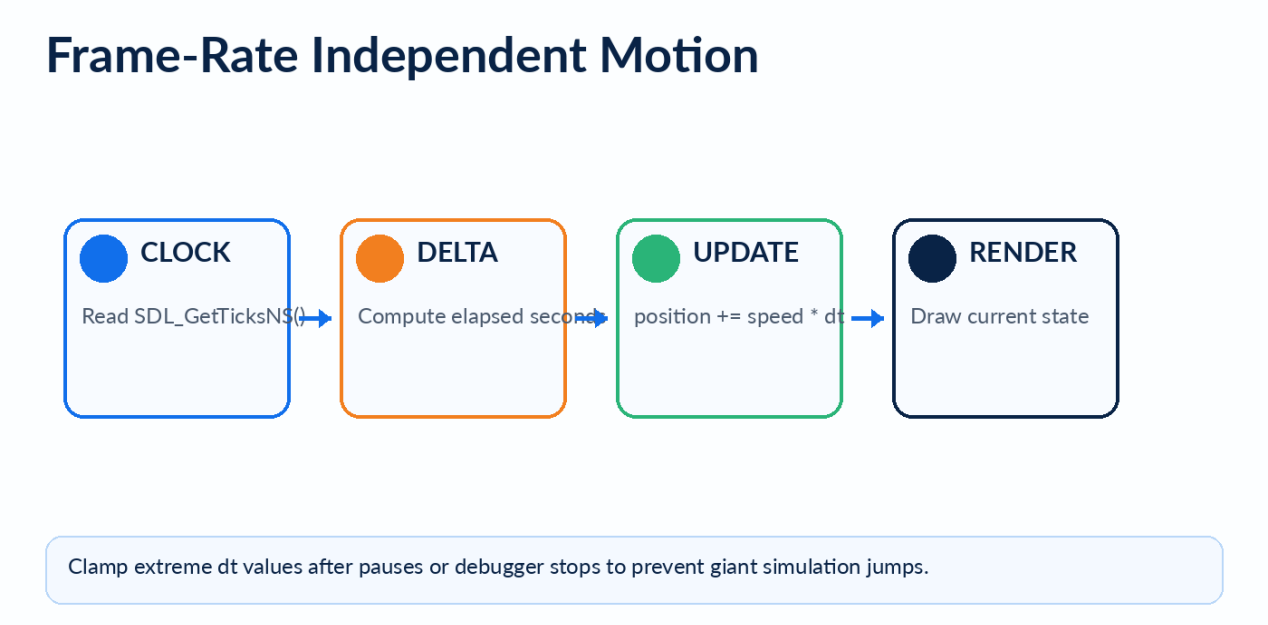
A sprite is simply a texture region rendered at a destination rectangle. A sprite sheet packs many frames into one texture. Animation is therefore a bookkeeping problem: choose the source rectangle, advance it at the right time, and keep movement state separate from animation state.

Core Mental Model

- Source rectangles select a region from the texture.
- Destination rectangles choose where and how large the frame appears.
- Animation speed is time-based, not frame-count-based.
- Movement state chooses the clip; the animation system advances the clip.

Key APIs / Tools

- SDL_RenderTexture
- SDL_RenderTextureRotated
- SDL_FRect
- SDL_GetTicksNS
- SDL_SetTextureScaleMode



Visual model for Chapter 8.

CHAPTER 8 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_RenderTexture` — copies a texture region to the active rendering target.

`SDL_RenderTextureRotated` — is a central function, type, target, or configuration point for this chapter.

`SDL_FRect` — is a central function, type, target, or configuration point for this chapter.

`SDL_GetTicksNS` — is a central function, type, target, or configuration point for this chapter.

`SDL_SetTextureScaleMode` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 8 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
static Uint64 previous_ns = 0;
Uint64 now_ns = SDL_GetTicksNS();
if (previous_ns == 0) previous_ns = now_ns;

float dt = (float)(now_ns - previous_ns) / 1000000000.0f;
previous_ns = now_ns;
if (dt > 0.05f) dt = 0.05f;   // clamp debugger/pause spikes

player_x += player_speed * dt;
```

What to Notice

- Load one sprite-sheet texture.
- Describe clips as first frame, frame count, and seconds per frame.
- Accumulate delta time.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 8 — WORKFLOW

The Reliable Step-by-Step Workflow

36. Load one sprite-sheet texture.
37. Describe clips as first frame, frame count, and seconds per frame.
38. Accumulate delta time.
39. Advance frames while enough time has elapsed.
40. Render the selected source rectangle.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 8 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
int frame = (int)(animation_time / seconds_per_frame) % frame_count;
int col = frame % columns;
int row = frame / columns;
SDL_FRect src = { col * frame_w, row * frame_h, frame_w, frame_h };
SDL_RenderTexture(renderer, sheet, &src, &dst);
```

What to Notice

- Load one sprite-sheet texture.
- Describe clips as first frame, frame count, and seconds per frame.
- Accumulate delta time.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 8 — FAILURE MODES

Mistakes That Waste the Most Time

1. Advancing one frame per render makes animation speed depend on machine speed

Advancing one frame per render makes animation speed depend on machine speed.

2. Mixing physics movement and animation indexing creates hard-to-fix state bugs

Mixing physics movement and animation indexing creates hard-to-fix state bugs.

3. For pixel art, linear filtering may blur the sheet

For pixel art, linear filtering may blur the sheet.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 8 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain source rectangles select a region from the texture..
- I can explain destination rectangles choose where and how large the frame appears..
- I can explain animation speed is time-based, not frame-count-based..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Create idle, walk, and one-shot jump clips from a synthetic sprite sheet. Switch state with keyboard input and keep animation speed stable while resizing the window.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 8 — DEEPER DESIGN

Going One Layer Deeper Without Losing Simplicity

A robust animation component owns only visual state: clip, frame index, elapsed time, direction, and optional playback rate. Higher-level gameplay code decides whether the character is idle, walking, jumping, or attacking. That separation lets the same animation system work in games, UI transitions, simulations, and visualization tools.

Design Boundary

Create small helpers when they remove repeated ownership or repeated error checks. Avoid wrappers that rename every SDL function without adding a real policy. The official SDL documentation should remain easy to map onto your codebase.

Measure, Do Not Guess

- Time real workloads, not empty loops.
- Profile asset loading separately from frame rendering.
- Test on the weakest target you intend to support.
- Keep debug overlays available in development builds.

Good Abstraction An abstraction is earning its place when it makes ownership, lifetime, or intent clearer — not merely when it reduces the number of SDL names visible in your code.

CHAPTER 9 — CORE GRAPHICS

Text Rendering and Simple HUD Design

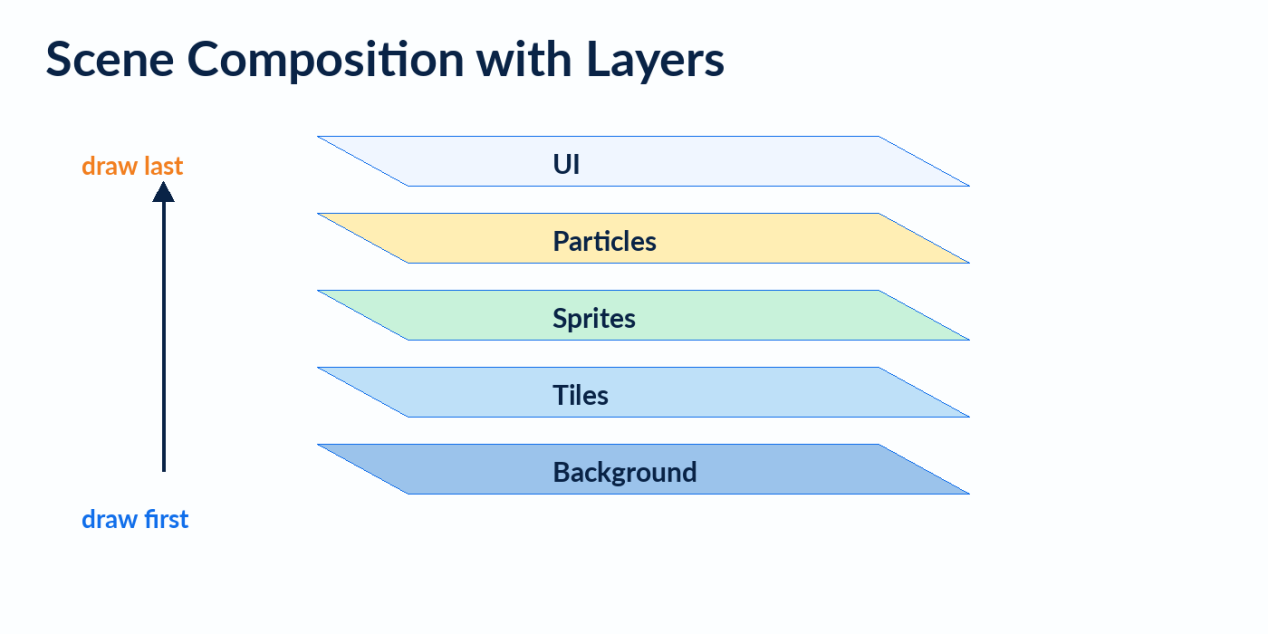
SDL_ttf 3 adds modern text objects and renderer text engines on top of FreeType and HarfBuzz. The essential lesson is the same as with textures: open a font once, create reusable text objects, update them only when content changes, and draw them after the world so the interface remains readable.

Core Mental Model

- TTF_Init is required before SDL_ttf use.
- TTF_CreateRendererTextEngine binds text drawing to an SDL_Renderer.
- TTF_Text objects can be recolored and updated without rebuilding your entire UI architecture.
- Fonts are assets with licenses; redistribution rights matter.

Key APIs / Tools

- TTF_Init
- TTF_OpenFont
- TTF_CreateRendererTextEngine
- TTF_CreateText
- TTF_DrawRendererText



Visual model for Chapter 9.

CHAPTER 9 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`TTF_Init` — is a central function, type, target, or configuration point for this chapter.

`TTF_OpenFont` — is a central function, type, target, or configuration point for this chapter.

`TTF_CreateRendererTextEngine` — is a central function, type, target, or configuration point for this chapter.

`TTF_CreateText` — is a central function, type, target, or configuration point for this chapter.

`TTF_DrawRendererText` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 9 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
#include <SDL3_ttf/SDL_ttf.h>

if (!TTF_Init()) {
    SDL_Log("TTF_Init: %s", SDL_GetError());
}
TTF_Font *font = TTF_OpenFont("assets/Inter-Regular.ttf", 28.0f);
TTF_TextEngine *engine = TTF_CreateRendererTextEngine(renderer);
TTF_Text *text = TTF_CreateText(engine, font, "SDL3 is clear!", 0);
TTF_SetTextColor(text, 240, 246, 255, 255);

TTF_DrawRendererText(text, 32.0f, 28.0f);

TTF_DestroyText(text);
TTF_DestroyRendererTextEngine(engine);
TTF_CloseFont(font);
TTF_Quit();
```

What to Notice

- Initialize SDL_ttf.
- Open the font once.
- Create a renderer text engine.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 9 — WORKFLOW

The Reliable Step-by-Step Workflow

41. Initialize SDL_ttf.
42. Open the font once.
43. Create a renderer text engine.
44. Create/update text objects when strings change.
45. Draw text during the UI phase and destroy objects on shutdown.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 9 — WORKING CODE

A Minimal Pattern You Can Build On

Example

CODE • COPY-READY

```
TTF_SetTextColor(text, 255, 210, 60, 255);
TTF_DrawRendererText(text, 24.0f, 20.0f);
```

What to Notice

- Initialize SDL_ttf.
- Open the font once.
- Create a renderer text engine.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 9 — FAILURE MODES

Mistakes That Waste the Most Time

1. Recreating font objects every frame wastes time

Recreating font objects every frame wastes time.

2. Tiny text rendered into a heavily scaled logical canvas can look soft

Tiny text rendered into a heavily scaled logical canvas can look soft.

3. Do not redistribute commercial fonts without permission

Do not redistribute commercial fonts without permission.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 9 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain tTF_Init is required before SDL_ttf use..
- I can explain tTF_CreateRendererTextEngine binds text drawing to an SDL_Renderer..
- I can explain tTF_Text objects can be recolored and updated without rebuilding your entire UI architecture..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Create a HUD containing FPS, mouse coordinates, and an object count. Update only the strings that actually change.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 10 — CORE GRAPHICS

Render Targets, Layers, and Scene Composition

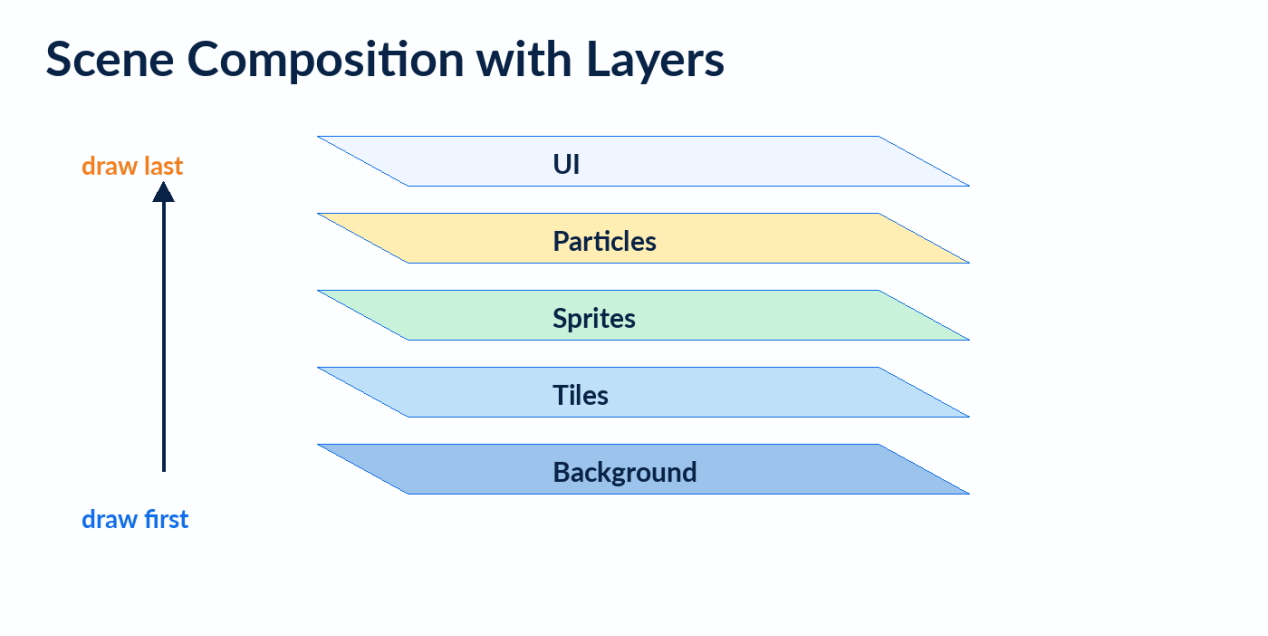
A render target is a texture that can receive draw commands. This single feature enables scene caching, mini-maps, post-processing-style workflows, low-resolution rendering, UI composition, and clean separation between layers.

Core Mental Model

- Target textures must be created with SDL_TEXTUREACCESS_TARGET.
- SDL_SetRenderTarget(renderer, texture) redirects subsequent draw commands.
- Pass NULL to restore the window as the render target.
- Viewport, scale, clip, and logical presentation state are per render target.

Key APIs / Tools

- SDL_CreateTexture
- SDL_TEXTUREACCESS_TARGET
- SDL_SetRenderTarget
- SDL_GetRenderTarget
- SDL_RenderTexture



Visual model for Chapter 10.

CHAPTER 10 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_CreateTexture` — is a central function, type, target, or configuration point for this chapter.

`SDL_TEXTUREACCESS_TARGET` — is a central function, type, target, or configuration point for this chapter.

`SDL_SetRenderTarget` — is a central function, type, target, or configuration point for this chapter.

`SDL_GetRenderTarget` — is a central function, type, target, or configuration point for this chapter.

`SDL_RenderTexture` — copies a texture region to the active rendering target.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 10 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_Texture *scene = SDL_CreateTexture(
    renderer, SDL_PIXELFORMAT_RGBA8888,
    SDL_TEXTUREACCESS_TARGET, 640, 360);

SDL_SetRenderTarget(renderer, scene);
SDL_SetRenderDrawColor(renderer, 18, 24, 38, 255);
SDL_RenderClear(renderer);
DrawWorld(renderer);
DrawParticles(renderer);

SDL_SetRenderTarget(renderer, NULL);
SDL_RenderTexture(renderer, scene, NULL, NULL);
DrawUI(renderer);
SDL_RenderPresent(renderer);
```

What to Notice

- Create the target once.

- Set the target and draw one or more layers.
- Restore the target to NULL.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 10 — WORKFLOW

The Reliable Step-by-Step Workflow

46. Create the target once.
47. Set the target and draw one or more layers.
48. Restore the target to NULL.
49. Draw the composed target to the window.
50. Draw final UI and present.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 10 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_Texture *old = SDL_GetRenderTarget(renderer);
SDL_SetRenderTarget(renderer, effect_target);
DrawEffectLayer(renderer);
SDL_SetRenderTarget(renderer, old);
```

What to Notice

- Create the target once.
- Set the target and draw one or more layers.
- Restore the target to NULL.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 10 — FAILURE MODES

Mistakes That Waste the Most Time

1. Forgetting to restore NULL means your “final” draw may still go into the texture

Forgetting to restore NULL means your “final” draw may still go into the texture.

2. Render-target dimensions should be chosen deliberately

Render-target dimensions should be chosen deliberately.

3. Target textures consume GPU memory; do not create them casually every frame

Target textures consume GPU memory; do not create them casually every frame.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 10 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain target textures must be created with SDL_TEXTUREACCESS_TARGET..
- I can explain sDL_SetRenderTarget(renderer, texture) redirects subsequent draw commands..
- I can explain pass NULL to restore the window as the render target..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Render a static background into one target texture, dynamic objects directly each frame, and a mini-map into a second target.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 11 — CORE GRAPHICS

Pixel Precision, Scaling, and High-DPI Windows

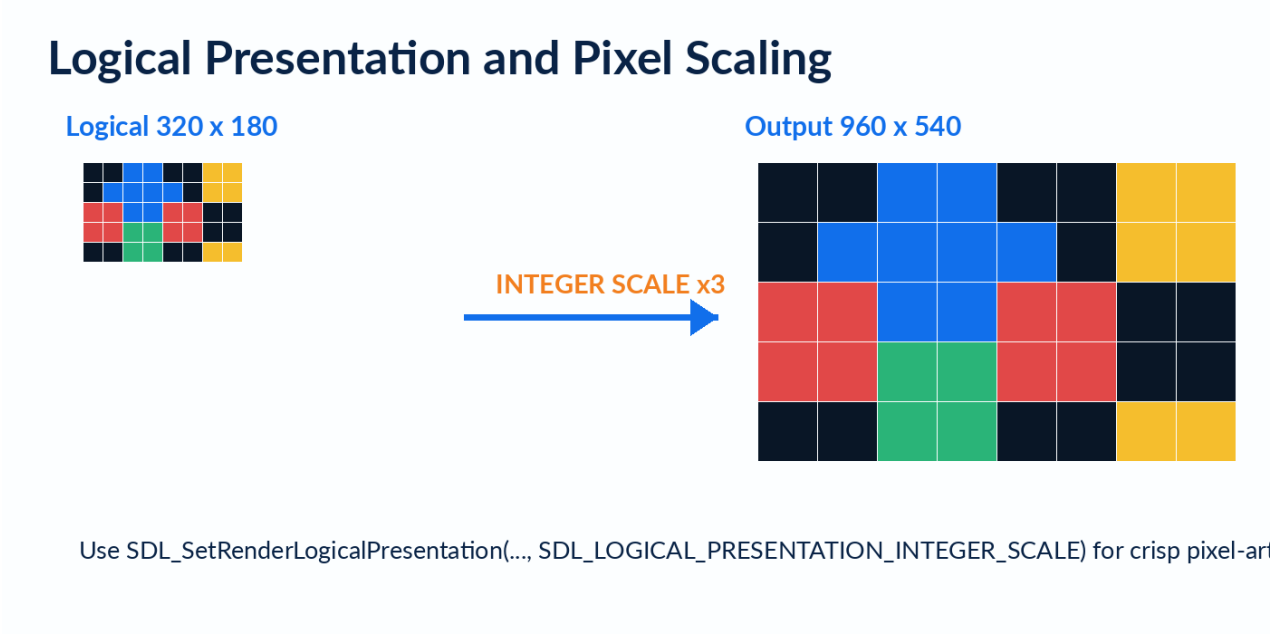
Modern displays vary wildly in pixel density and window size. SDL3 separates the actual output size from a logical presentation size, letting a program design at a stable resolution while SDL maps that coordinate space to the real output.

Core Mental Model

- High pixel density does not require you to hard-code physical pixels.
- Logical presentation can stretch, letterbox, overscan, or integer-scale.
- Pixel art benefits from nearest or pixel-art scale modes.
- Input coordinates can be converted to render coordinates when logical presentation is active.

Key APIs / Tools

- SDL_SetRenderLogicalPresentation
- SDL_LOGICAL_PRESENTATION_INTEGER_SCALE
- SDL_SetTextureScaleMode
- SDL_SCALEMODE_NEAREST
- SDL_SCALEMODE_PIXELART



Visual model for Chapter 11.

CHAPTER 11 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_SetRenderLogicalPresentation` — defines a device-independent render resolution and scaling policy.

`SDL_LOGICAL_PRESENTATION_INTEGER_SCALE` — is a central function, type, target, or configuration point for this chapter.

`SDL_SetTextureScaleMode` — is a central function, type, target, or configuration point for this chapter.

`SDL_SCALEMODE_NEAREST` — is a central function, type, target, or configuration point for this chapter.

`SDL_SCALEMODE_PIXELART` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 11 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
if (!SDL_SetRenderLogicalPresentation(
        renderer, 320, 180,
        SDL_LOGICAL_PRESENTATION_INTEGER_SCALE)) {
    SDL_Log("logical presentation: %s", SDL_GetError());
}

SDL_SetDefaultTextureScaleMode(renderer, SDL_SCALEMODE_PIXELART);
// For an individual texture:
SDL_SetTextureScaleMode(texture, SDL_SCALEMODE_NEAREST);
```

What to Notice

- Choose the logical canvas intentionally.
- Enable high-pixel-density window support.
- Set logical presentation on the renderer.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 11 — WORKFLOW

The Reliable Step-by-Step Workflow

51. Choose the logical canvas intentionally.
52. Enable high-pixel-density window support.
53. Set logical presentation on the renderer.
54. Choose texture scale modes based on art style.
55. Test multiple DPI and aspect ratios.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 11 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_ConvertEventToRenderCoordinates(renderer, &event);
// event coordinates now match the renderer logical space.
```

What to Notice

- Choose the logical canvas intentionally.
- Enable high-pixel-density window support.
- Set logical presentation on the renderer.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 11 — FAILURE MODES

Mistakes That Waste the Most Time

1. Invented APIs such as SDL_SetWindowLogicalSize are not the SDL3 renderer solution

Invented APIs such as SDL_SetWindowLogicalSize are not the SDL3 renderer solution.

2. Default linear texture scaling can blur pixel art

Default linear texture scaling can blur pixel art.

3. UI text may need a different scaling strategy than world pixels

UI text may need a different scaling strategy than world pixels.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 11 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain high pixel density does not require you to hard-code physical pixels..
- I can explain logical presentation can stretch, letterbox, overscan, or integer-scale..
- I can explain pixel art benefits from nearest or pixel-art scale modes..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Render the same 320x180 scene into 640x360, 960x540, and a non-matching aspect ratio. Compare stretch, letterbox, and integer-scale modes.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 12 — INPUT & INTERACTION

The Event System Explained

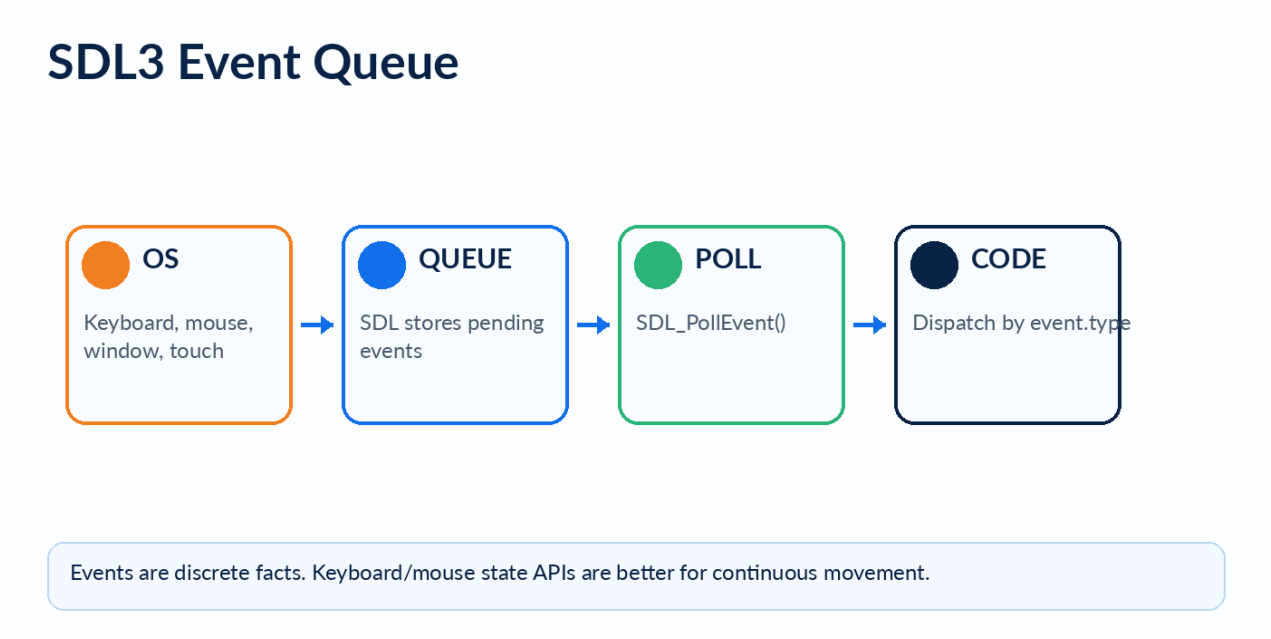
SDL’s event queue is the cross-platform stream of things that happened: keys pressed, mouse moved, window resized, controller attached, text entered, files dropped, touch gestures, and more. Events are ideal for discrete transitions and lifecycle changes.

Core Mental Model

- SDL_Event is a union; event.type tells you which member is valid.
- SDL_PollEvent removes one queued event at a time.
- Event timestamps are useful for diagnostics and input analysis.
- You can combine events with state-query APIs for a clean input system.

Key APIs / Tools

- SDL_Event
- SDL_PollEvent
- SDL_EventType
- SDL_EVENT_KEY_DOWN
- SDL_EVENT_WINDOW_RESIZED



Visual model for Chapter 12.

CHAPTER 12 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_Event` — is a central function, type, target, or configuration point for this chapter.

`SDL_PollEvent` — removes the next pending event and writes it into SDL_Event.

`SDL_EventType` — is a central function, type, target, or configuration point for this chapter.

`SDL_EVENT_KEY_DOWN` — is a central function, type, target, or configuration point for this chapter.

`SDL_EVENT_WINDOW_RESIZED` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 12 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_Event e;
while (SDL_PollEvent(&e)) {
    switch (e.type) {
    case SDL_EVENT_QUIT:
        running = false;
        break;
    case SDL_EVENT_KEY_DOWN:
        if (e.key.key == SDLK_ESCAPE) running = false;
        break;
    case SDL_EVENT_WINDOW_RESIZED:
        SDL_Log("window: %dx%d", e.window.data1, e.window.data2);
        break;
    default:
        break;
    }
}
```

What to Notice

- Poll until the queue is empty.
- Switch on event.type.
- Handle only the event-specific union member that matches the type.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 12 — WORKFLOW

The Reliable Step-by-Step Workflow

56. Poll until the queue is empty.
57. Switch on event.type.
58. Handle only the event-specific union member that matches the type.
59. Translate events into application actions or state changes.
60. Keep rendering independent.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 12 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_Log("event type = %u, timestamp = %llu",
        (unsigned)e.type,
        (unsigned long long)e.common.timestamp);
```

What to Notice

- Poll until the queue is empty.
- Switch on event.type.
- Handle only the event-specific union member that matches the type.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 12 — FAILURE MODES

Mistakes That Waste the Most Time

1. SDL2 names like SDL_QUIT and SDL_KEYDOWN are not the SDL3 event constants

SDL2 names like SDL_QUIT and SDL_KEYDOWN are not the SDL3 event constants.

2. Reading the wrong union member for an event type is logically invalid

Reading the wrong union member for an event type is logically invalid.

3. A single key press event is not a good representation of “key held for movement

A single key press event is not a good representation of “key held for movement.”

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 12 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain sDL_Event is a union; event.type tells you which member is valid..
- I can explain sDL_PollEvent removes one queued event at a time..
- I can explain event timestamps are useful for diagnostics and input analysis..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Write an event logger that prints event types, key codes, mouse coordinates, and window sizes for thirty seconds.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 13 — INPUT & INTERACTION

Keyboard and Mouse Input

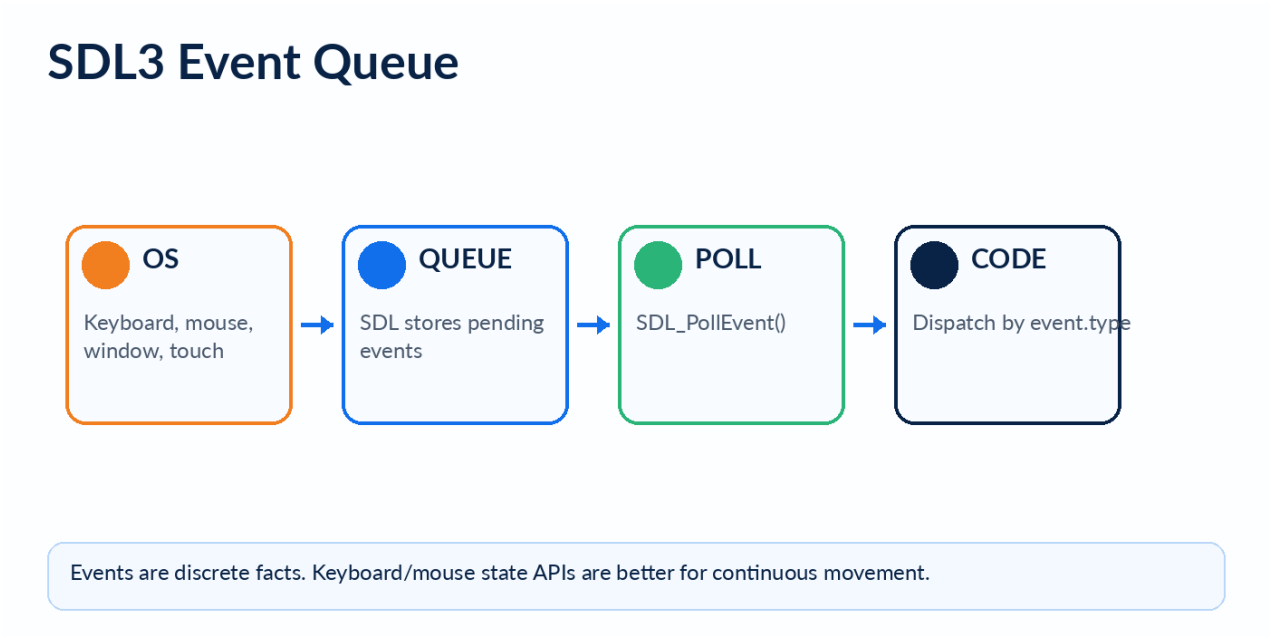
For continuous movement and interactive tools, synchronous state queries are often clearer than individual events. SDL_GetKeyboardState returns an internal boolean array indexed by SDL_Scancode. SDL_GetMouseState returns window-relative floating-point coordinates and a button bitmask.

Core Mental Model

- Scancodes describe physical key positions independent of keyboard layout.
- Keycodes describe layout-aware virtual keys and are useful for commands/text-like shortcuts.
- Mouse state is cached from the event pump.
- Relative mouse mode is valuable for cameras and first-person-style controls.

Key APIs / Tools

- SDL_GetKeyboardState
- SDL_GetMouseState
- SDL_GetRelativeMouseState
- SDL_BUTTON_LMASK
- SDL_SCANCODE_W



Visual model for Chapter 13.

CHAPTER 13 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_GetKeyboardState` — is a central function, type, target, or configuration point for this chapter.

`SDL_GetMouseState` — is a central function, type, target, or configuration point for this chapter.

`SDL_GetRelativeMouseState` — is a central function, type, target, or configuration point for this chapter.

`SDL_BUTTON_LMASK` — is a central function, type, target, or configuration point for this chapter.

`SDL_SCANCODE_W` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 13 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
const bool *keys = SDL_GetKeyboardState(NULL);
float mx = 0.0f, my = 0.0f;
SDL_MouseButtonFlags buttons = SDL_GetMouseState(&mx, &my);

float dx = 0.0f, dy = 0.0f;
if (keys[SDL_SCANCODE_A]) dx -= 1.0f;
if (keys[SDL_SCANCODE_D]) dx += 1.0f;
if (keys[SDL_SCANCODE_W]) dy -= 1.0f;
if (keys[SDL_SCANCODE_S]) dy += 1.0f;

bool left_down = (buttons & SDL_BUTTON_LMASK) != 0;
```

What to Notice

- Pump/process events first.
- Read keyboard state once per frame.
- Read mouse position and button mask.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 13 — WORKFLOW

The Reliable Step-by-Step Workflow

61. Pump/process events first.
62. Read keyboard state once per frame.
63. Read mouse position and button mask.
64. Convert raw input into high-level actions.
65. Update the model; then render.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 13 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
float relx = 0.0f, rely = 0.0f;
SDL_GetRelativeMouseState(&relx, &rely);
camera_yaw += relx * sensitivity;
```

What to Notice

- Pump/process events first.
- Read keyboard state once per frame.
- Read mouse position and button mask.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 13 — FAILURE MODES

Mistakes That Waste the Most Time

1. Using keycodes as indexes into SDL_GetKeyboardState is wrong; use scancodes

Using keycodes as indexes into SDL_GetKeyboardState is wrong; use scancodes.

2. Global mouse state is more expensive and uses desktop coordinates

Global mouse state is more expensive and uses desktop coordinates.

3. Mouse coordinates need conversion when renderer logical presentation is active

Mouse coordinates need conversion when renderer logical presentation is active.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 13 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain scancodes describe physical key positions independent of keyboard layout..
- I can explain keycodes describe layout-aware virtual keys and are useful for commands/text-like shortcuts..
- I can explain mouse state is cached from the event pump..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Build a draggable rectangle: WASD moves it, left mouse drag repositions it, right mouse toggles its color, and Escape quits.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 14 — INPUT & INTERACTION

Touch Input and Game Controllers

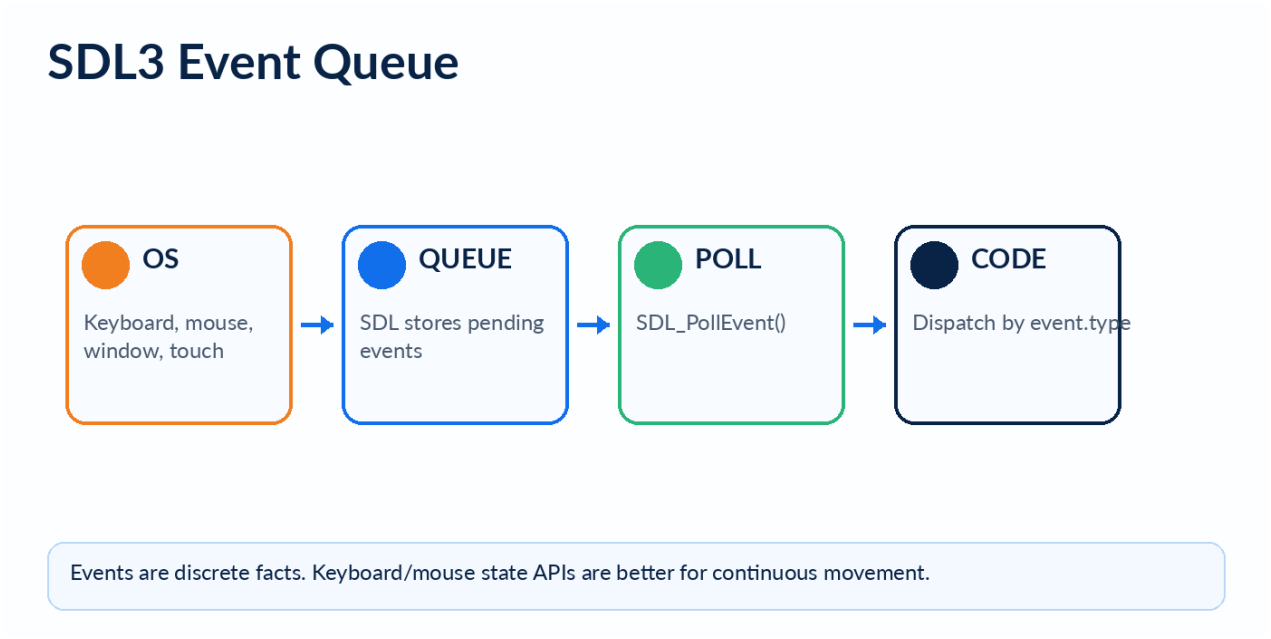
SDL3 normalizes touch and gamepad input across platforms. Touch arrives primarily through finger events. Gamepads use a higher-level standardized mapping so your code talks about “south button” and “left stick” instead of device-specific button numbers.

Core Mental Model

- Finger coordinates are typically normalized and belong to a touch device.
- Touch may synthesize mouse events by default.
- SDL gamepads use standard semantic locations for buttons and axes.
- Devices can appear and disappear at runtime.

Key APIs / Tools

- SDL_EVENT_FINGER_DOWN
- SDL_EVENT_FINGER_MOTION
- SDL_GetGamepads
- SDL_OpenGamepad
- SDL_GetGamepadButton



Visual model for Chapter 14.

CHAPTER 14 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

SDL_EVENT_FINGER_DOWN — is a central function, type, target, or configuration point for this chapter.

SDL_EVENT_FINGER_MOTION — is a central function, type, target, or configuration point for this chapter.

SDL_GetGamepads — is a central function, type, target, or configuration point for this chapter.

SDL_OpenGamepad — is a central function, type, target, or configuration point for this chapter.

SDL_GetGamepadButton — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 14 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
int count = 0;
SDL_JoystickID *ids = SDL_GetGamepads(&count);
SDL_Gamepad *pad = NULL;
if (ids && count > 0) pad = SDL_OpenGamepad(ids[0]);
SDL_free(ids);

if (pad) {
    bool jump = SDL_GetGamepadButton(pad, SDL_GAMEPAD_BUTTON_SOUTH);
    Sint16 x = SDL_GetGamepadAxis(pad, SDL_GAMEPAD_AXIS_LEFTX);
    // normalize x to approximately -1..1 before movement
    (void)jump; (void)x;
    SDL_CloseGamepad(pad);
}
```

What to Notice

- Initialize the gamepad subsystem when needed.
- Enumerate and open available gamepads.

- Handle add/remove device events.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 14 — WORKFLOW

The Reliable Step-by-Step Workflow

66. Initialize the gamepad subsystem when needed.
67. Enumerate and open available gamepads.
68. Handle add/remove device events.
69. Read standardized buttons and axes.
70. Close gamepads explicitly.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 14 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
if (e.type == SDL_EVENT_FINGER_DOWN ||
    e.type == SDL_EVENT_FINGER_MOTION) {
    float nx = e.tfinger.x;
    float ny = e.tfinger.y;
    // map normalized touch coordinates to your logical canvas
}
```

What to Notice

- Initialize the gamepad subsystem when needed.
- Enumerate and open available gamepads.
- Handle add/remove device events.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 14 — FAILURE MODES

Mistakes That Waste the Most Time

1. Do not assume controller button labels are the same across vendors

Do not assume controller button labels are the same across vendors.

2. Do not keep using a gamepad pointer after the device is removed

Do not keep using a gamepad pointer after the device is removed.

3. If you process touch and synthetic mouse events together, one gesture may be handled twice

If you process touch and synthetic mouse events together, one gesture may be handled twice.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 14 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain finger coordinates are typically normalized and belong to a touch device..
- I can explain touch may synthesize mouse events by default..
- I can explain sDL gamepads use standard semantic locations for buttons and axes..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Create one on-screen square controllable by mouse, touch drag, keyboard, or the gamepad left stick, all through the same high-level MoveAction structure.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 15 — INPUT & INTERACTION

Timing, Delta Time, and Smooth Movement

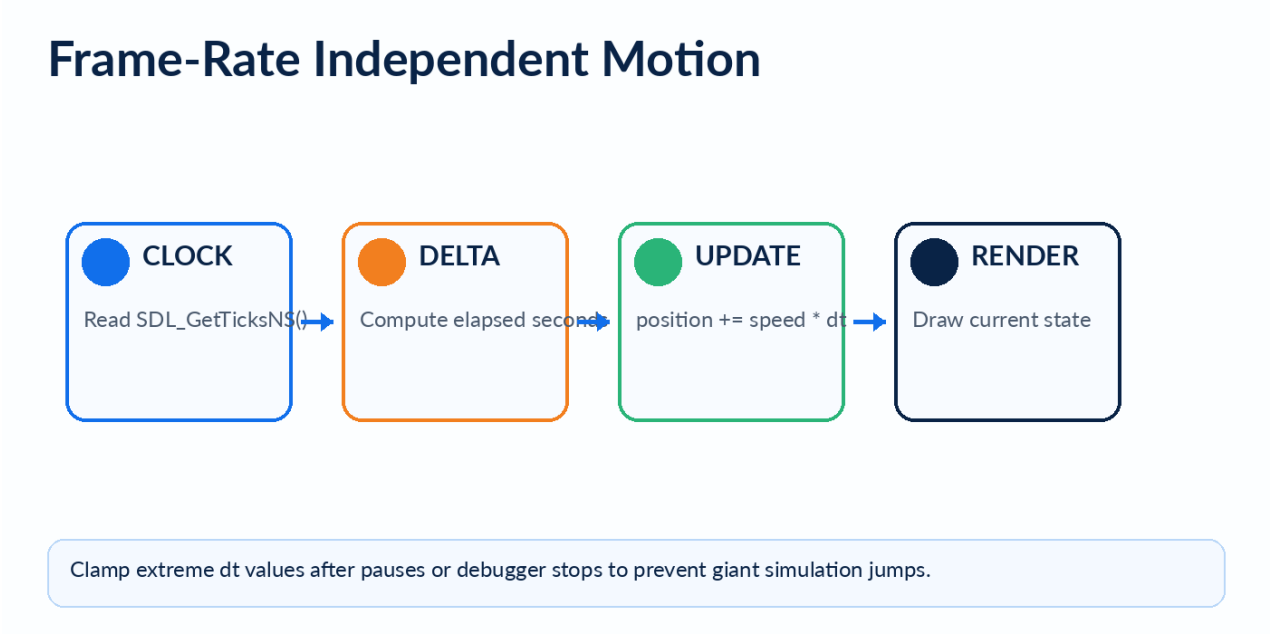
Smooth graphics are not created by “sleeping 16 ms.” They come from separating simulation speed from frame rate. Delta time measures how much real time elapsed since the previous update so movement can be expressed in units per second.

Core Mental Model

- SDL_GetTicksNS gives a monotonic nanosecond-scale application timer.
- Convert nanoseconds to seconds for intuitive movement equations.
- Clamp giant delta values caused by pauses or debugger stops.
- Fixed-step simulation can be layered on top when determinism matters.

Key APIs / Tools

- SDL_GetTicksNS
- SDL_GetTicks
- SDL_DelayNS
- SDL_DelayPrecise



Visual model for Chapter 15.

CHAPTER 15 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

SDL_GetTicksNS — is a central function, type, target, or configuration point for this chapter.

SDL_GetTicks — is a central function, type, target, or configuration point for this chapter.

SDL_DelayNS — is a central function, type, target, or configuration point for this chapter.

SDL_DelayPrecise — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 15 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
static Uint64 previous_ns = 0;
Uint64 now_ns = SDL_GetTicksNS();
if (previous_ns == 0) previous_ns = now_ns;

float dt = (float)(now_ns - previous_ns) / 1000000000.0f;
previous_ns = now_ns;
if (dt > 0.05f) dt = 0.05f;   // clamp debugger/pause spikes

player_x += player_speed * dt;
```

What to Notice

- Read the current time.
- Compute dt = now - previous.
- Clamp pathological spikes.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 15 — WORKFLOW

The Reliable Step-by-Step Workflow

71. Read the current time.
72. Compute dt = now - previous.
73. Clamp pathological spikes.
74. Update motion with speed * dt.
75. Render the resulting state.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 15 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```
const float fixed_dt = 1.0f / 120.0f;
accumulator += dt;
while (accumulator >= fixed_dt) {
    Simulate(fixed_dt);
    accumulator -= fixed_dt;
}
```

What to Notice

- Read the current time.
- Compute dt = now - previous.
- Clamp pathological spikes.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 15 — FAILURE MODES

Mistakes That Waste the Most Time

1. Frame-based increments move faster on faster machines

Frame-based increments move faster on faster machines.

2. Sleeping is not a substitute for measuring elapsed time

Sleeping is not a substitute for measuring elapsed time.

3. An uncapped render loop may waste power if your application does not need maximum FPS

An uncapped render loop may waste power if your application does not need maximum FPS.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 15 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain sDL_GetTicksNS gives a monotonic nanosecond-scale application timer..
- I can explain convert nanoseconds to seconds for intuitive movement equations..
- I can explain clamp giant delta values caused by pauses or debugger stops..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Move an object at exactly 200 logical pixels per second. Add an FPS counter and compare motion with vsync on and off.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 16 — AUDIO & SHIPPING

Audio Playback, Streams, and Practical Sound

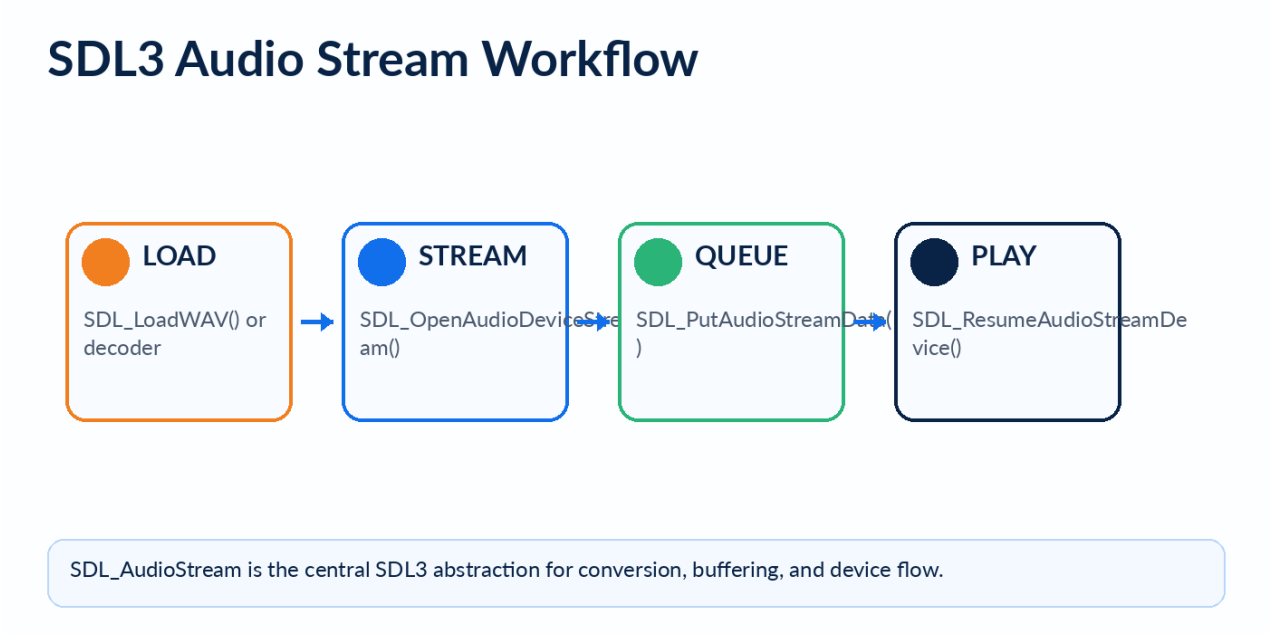
SDL3 audio is substantially different from SDL2. Audio streams are now central: they buffer, convert formats, resample, remap channels, and bind to devices. For simple playback, SDL_OpenAudioDeviceStream creates and binds the stream in one call.

Core Mental Model

- SDL_AudioStream is the core conversion and queue abstraction.
- SDL_OpenAudioDeviceStream returns a stream associated with a device.
- That convenience function starts paused; resume it for playback.
- SDL_LoadWAV is a convenient loader for PCM WAV data.

Key APIs / Tools

- SDL_OpenAudioDeviceStream
- SDL_PutAudioStreamData
- SDL_ResumeAudioStreamDevice
- SDL_DestroyAudioStream
- SDL_LoadWAV



Visual model for Chapter 16.

CHAPTER 16 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

SDL_OpenAudioDeviceStream — opens an audio device, creates a stream, and binds them for straightforward playback.

SDL_PutAudioStreamData — is a central function, type, target, or configuration point for this chapter.

SDL_ResumeAudioStreamDevice — is a central function, type, target, or configuration point for this chapter.

SDL_DestroyAudioStream — is a central function, type, target, or configuration point for this chapter.

SDL_LoadWAV — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 16 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_AudioSpec spec = {
    .format = SDL_AUDIO_F32,
    .channels = 2,
    .freq = 48000
};
SDL_AudioStream *stream = SDL_OpenAudioDeviceStream(
    SDL_AUDIO_DEVICE_DEFAULT_PLAYBACK, &spec, NULL, NULL);
if (!stream) {
    SDL_Log("audio stream: %s", SDL_GetError());
    return false;
}

SDL_PutAudioStreamData(stream, pcm, pcm_bytes);
SDL_ResumeAudioStreamDevice(stream);
// ... keep application running ...
SDL_DestroyAudioStream(stream);
```

What to Notice

- Load or generate PCM audio.
- Open an output stream with the format you will feed it.
- Queue audio bytes.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 16 — WORKFLOW

The Reliable Step-by-Step Workflow

76. Load or generate PCM audio.
77. Open an output stream with the format you will feed it.
78. Queue audio bytes.
79. Resume the stream device.
80. Destroy the stream to free the stream and close its convenience-opened device.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 16 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
SDL_AudioSpec wav_spec;
Uint8 *wav_data = NULL;
Uint32 wav_len = 0;
if (!SDL_LoadWAV("assets/click.wav", &wav_spec, &wav_data, &wav_len)) {
    SDL_Log("SDL_LoadWAV: %s", SDL_GetError());
}
// Feed wav_data to an audio stream configured for wav_spec.
SDL_free(wav_data);
```

What to Notice

- Load or generate PCM audio.
- Open an output stream with the format you will feed it.
- Queue audio bytes.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 16 — FAILURE MODES

Mistakes That Waste the Most Time

1. Old SDL2 queue/callback examples can mislead SDL3 learners

Old SDL2 queue/callback examples can mislead SDL3 learners.

2. Buffer starvation causes clicks and gaps

Buffer starvation causes clicks and gaps.

3. Huge buffers increase perceived latency

Huge buffers increase perceived latency.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 16 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain sDL_AudioStream is the core conversion and queue abstraction..
- I can explain sDL_OpenAudioDeviceStream returns a stream associated with a device..
- I can explain that convenience function starts paused; resume it for playback..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Load a click WAV and play it when Space is pressed. Then generate a 440 Hz sine wave in float stereo and queue it to a second demo stream.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 16 — DEEPER DESIGN

Going One Layer Deeper Without Losing Simplicity

For games and soundboards, build a tiny mixer layer above SDL_AudioStream instead of opening a new device for every sound. Decode assets ahead of time, mix or queue short effects with controlled gain, and keep the audio thread free from blocking file I/O. Measure latency on the real target machine.

Design Boundary

Create small helpers when they remove repeated ownership or repeated error checks. Avoid wrappers that rename every SDL function without adding a real policy. The official SDL documentation should remain easy to map onto your codebase.

Measure, Do Not Guess

- Time real workloads, not empty loops.

- Profile asset loading separately from frame rendering.
- Test on the weakest target you intend to support.
- Keep debug overlays available in development builds.

Good Abstraction An abstraction is earning its place when it makes ownership, lifetime, or intent clearer — not merely when it reduces the number of SDL names visible in your code.

CHAPTER 17 — PROJECT

Build a Small 2D Graphics Demo Step by Step

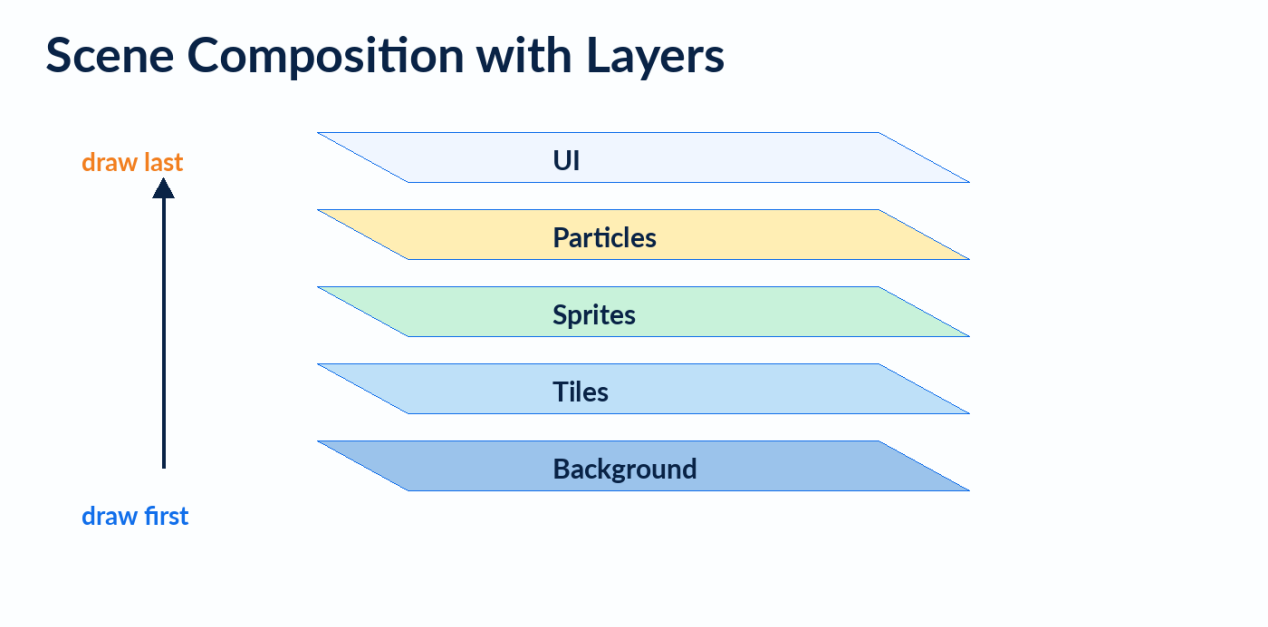
This chapter assembles the book into one small application: a resizable logical canvas with a background, moving object, sprite-like texture, input, text HUD, and sound. The point is not to build a full game; it is to show how simple modules cooperate without becoming tangled.

Core Mental Model

- App owns SDL lifetime and top-level resources.
- Input produces actions, not rendering calls.
- Update owns motion and animation state.
- Renderer draws layers in a stable order.
- Assets are loaded once and released once.

Key APIs / Tools

- SDL_CreateWindowAndRenderer
- SDL_SetRenderLogicalPresentation
- IMG_LoadTexture
- TTF_DrawRendererText
- SDL_OpenAudioDeviceStream



Visual model for Chapter 17.

CHAPTER 17 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_CreateWindowAndRenderer` — is a central function, type, target, or configuration point for this chapter.

`SDL_SetRenderLogicalPresentation` — defines a device-independent render resolution and scaling policy.

`IMG_LoadTexture` — is a central function, type, target, or configuration point for this chapter.

`TTF_DrawRendererText` — is a central function, type, target, or configuration point for this chapter.

`SDL_OpenAudioDeviceStream` — opens an audio device, creates a stream, and binds them for straightforward playback.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 17 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
typedef struct {
    SDL_Window *window;
    SDL_Renderer *renderer;
    bool running;
} App;

typedef struct {
    float x, y;
    float vx, vy;
    SDL_Texture *art;
} Scene;
```

What to Notice

- Create App and Scene structs.
- Load graphical and audio assets.
- Run event/input/update/render loop.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 17 — WORKFLOW

The Reliable Step-by-Step Workflow

81. Create App and Scene structs.
82. Load graphical and audio assets.
83. Run event/input/update/render loop.
84. Keep world and UI coordinate decisions explicit.
85. Shut down in reverse ownership order.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 17 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
while (app.running) {
    float dt = Clock_Tick(&clock);
    Input_Poll(&input, &app.running);
    Scene_Update(&scene, &input, dt);
    Scene_Render(&scene, app.renderer);
    Hud_Render(&hud);
```

```
    SDL_RenderPresent(app.renderer);
}
```

What to Notice

- Create App and Scene structs.
- Load graphical and audio assets.
- Run event/input/update/render loop.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 17 — FAILURE MODES

Mistakes That Waste the Most Time

1. A “god main

A “god main.c” becomes difficult to reason about.

2. Resource ownership must not be duplicated across modules

Resource ownership must not be duplicated across modules.

3. Relative file paths should be centralized

Relative file paths should be centralized.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 17 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain app owns SDL lifetime and top-level resources..
- I can explain input produces actions, not rendering calls..
- I can explain update owns motion and animation state..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Complete the demo, then add one feature: camera scrolling, a second animated object, or a pause overlay.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 17 — DEEPER DESIGN

Going One Layer Deeper Without Losing Simplicity

A small graphics framework should make the simple path obvious: create, update, draw, destroy. Do not hide SDL behind dozens of abstraction layers. Wrap repeated ownership and error handling, but keep SDL types visible enough that the official documentation remains directly useful.

Design Boundary

Create small helpers when they remove repeated ownership or repeated error checks. Avoid wrappers that rename every SDL function without adding a real policy. The official SDL documentation should remain easy to map onto your codebase.

Measure, Do Not Guess

- Time real workloads, not empty loops.
- Profile asset loading separately from frame rendering.
- Test on the weakest target you intend to support.
- Keep debug overlays available in development builds.

Good Abstraction An abstraction is earning its place when it makes ownership, lifetime, or intent clearer — not merely when it reduces the number of SDL names visible in your code.

CHAPTER 18 — AUDIO & SHIPPING

Debugging, Logging, and Common SDL3 Mistakes

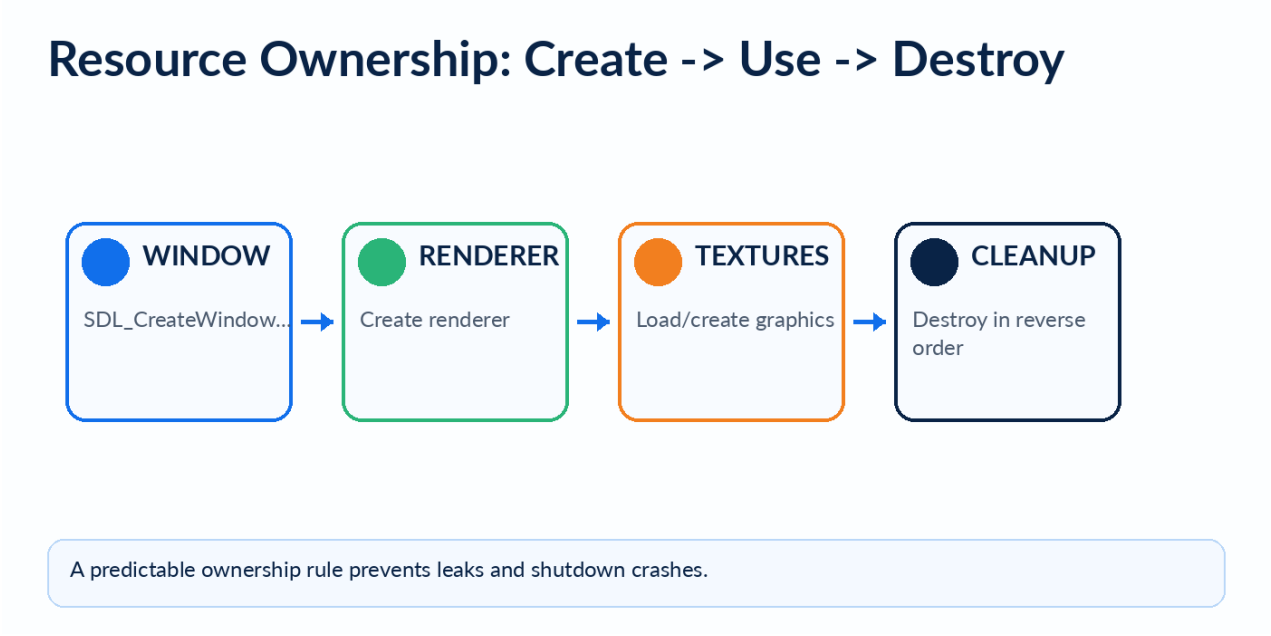
Graphics bugs are often state bugs: wrong target, wrong coordinates, wrong texture owner, wrong draw order, or a resource destroyed too early. A disciplined diagnostic method is faster than random edits: check every return value, log context, minimize the scene, and verify assumptions one layer at a time.

Core Mental Model

- SDL_GetError returns diagnostic text after many failed calls.
- SDL_Log is available without inventing a logging framework.
- Compiler warnings and sanitizers catch problems before visual symptoms appear.
- A minimal reproducible scene is a powerful graphics debugging tool.

Key APIs / Tools

- SDL_GetError
- SDL_Log
- SDL_LogError
- SDL_assert
- SDL_GetRenderTarget



Visual model for Chapter 18.

CHAPTER 18 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL_GetError` — is a central function, type, target, or configuration point for this chapter.

`SDL_Log` — is a central function, type, target, or configuration point for this chapter.

`SDL_LogError` — is a central function, type, target, or configuration point for this chapter.

`SDL_assert` — is a central function, type, target, or configuration point for this chapter.

`SDL_GetRenderTarget` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 18 — WORKING CODE

A Minimal Pattern You Can Build On

Example

CODE • COPY-READY

```
# GCC debug diagnostics
target_compile_options(app PRIVATE -Wall -Wextra -Wpedantic -Wconversion -Wshadow)
target_compile_options(app PRIVATE -fsanitize=address,undefined)
target_link_options(app PRIVATE -fsanitize=address,undefined)
```

What to Notice

- Check failure immediately.
- Log operation + asset path + relevant state.
- Reduce to a solid clear + one primitive.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 18 — WORKFLOW

The Reliable Step-by-Step Workflow

86. Check failure immediately.
87. Log operation + asset path + relevant state.
88. Reduce to a solid clear + one primitive.
89. Add layers back one at a time.
90. Use ASan/UBSan in a dedicated GCC profile.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 18 — WORKING CODE

A Minimal Pattern You Can Build On

Example

C23 • COPY-READY

```c
if (!texture) {
    SDL_LogError(SDL_LOG_CATEGORY_APPLICATION,
        "asset %s failed: %s", path, SDL_GetError());
    return false;
}
```

What to Notice

- Check failure immediately.
- Log operation + asset path + relevant state.
- Reduce to a solid clear + one primitive.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 18 — FAILURE MODES

Mistakes That Waste the Most Time

1. Ignoring a NULL texture and debugging the renderer afterward wastes time

Ignoring a NULL texture and debugging the renderer afterward wastes time.

2. Assuming the working directory is the source directory causes asset bugs

Assuming the working directory is the source directory causes asset bugs.

3. Overwriting the meaningful SDL error with later failing calls can hide the original problem

Overwriting the meaningful SDL error with later failing calls can hide the original problem.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 18 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain sDL_GetError returns diagnostic text after many failed calls..
- I can explain sDL_Log is available without inventing a logging framework..
- I can explain compiler warnings and sanitizers catch problems before visual symptoms appear..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Create a “sanitized” CMake profile with -fsanitize=address,undefined and intentionally trigger a use-after-free in a toy texture wrapper, then fix it.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 19 — AUDIO & SHIPPING

Dynamic Linking, Static Linking, and Release Builds

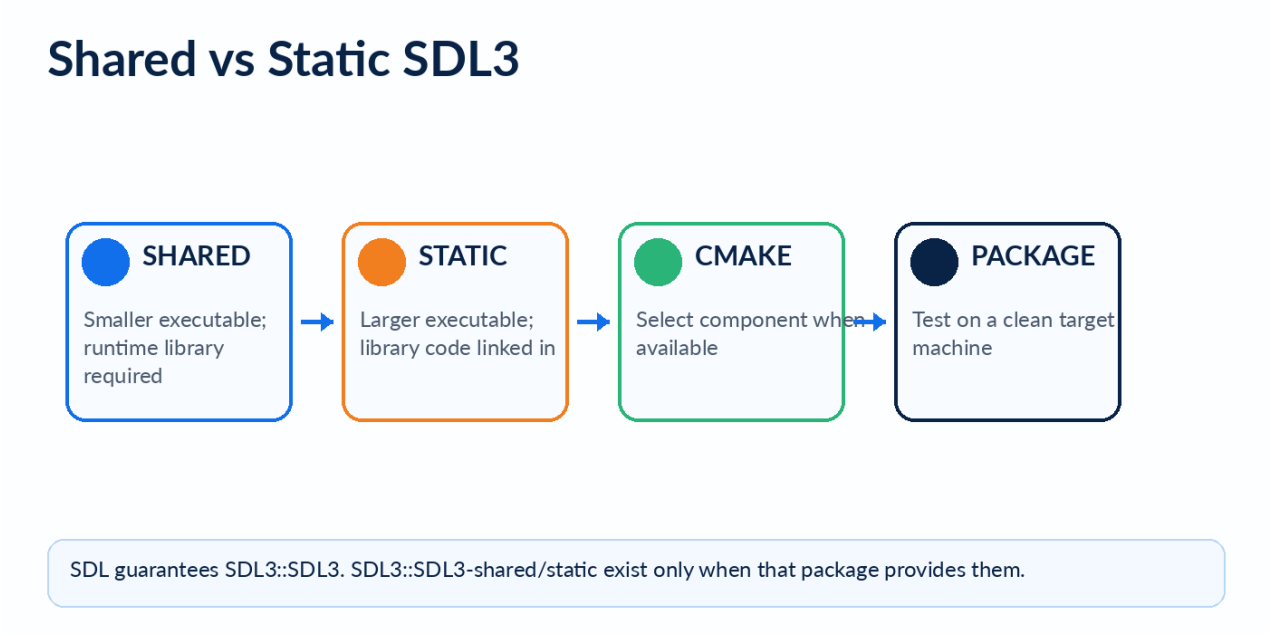
Linking strategy affects deployment, file size, updates, licensing, and which runtime files your users need. SDL’s official CMake integration guarantees SDL3::SDL3 and may expose explicit shared/static targets when the installed package provides them.

Core Mental Model

- Shared linking keeps SDL in a separate runtime library.
- Static linking copies library code into the final program.
- CMake target names are safer than hand-written -l flags.
- Static availability depends on how SDL was built and packaged.

Key APIs / Tools

- SDL3::SDL3
- SDL3::SDL3-shared
- SDL3::SDL3-static
- find_package(... COMPONENTS SDL3-shared)
- find_package(... COMPONENTS SDL3-static)



Visual model for Chapter 19.

CHAPTER 19 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`SDL3::SDL3` — is a central function, type, target, or configuration point for this chapter.

`SDL3::SDL3-shared` — is a central function, type, target, or configuration point for this chapter.

`SDL3::SDL3-static` — is a central function, type, target, or configuration point for this chapter.

`find_package(... COMPONENTS SDL3-shared)` — is a central function, type, target, or configuration point for this chapter.

`find_package(... COMPONENTS SDL3-static)` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 19 — WORKING CODE

A Minimal Pattern You Can Build On

Example

CMAKE • COPY-READY

```cmake
find_package(SDL3 CONFIG REQUIRED COMPONENTS SDL3-shared)
add_executable(app src/main.c)
target_link_libraries(app PRIVATE SDL3::SDL3)

if(WIN32 AND TARGET SDL3::SDL3-shared)
  add_custom_command(TARGET app POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy
      $<TARGET_FILE:SDL3::SDL3-shared>
      $<TARGET_FILE_DIR:app>)
endif()
```

What to Notice

- Choose shared or static intentionally.
- Configure CMake against the matching installed package.

- Build Release or RelWithDebInfo.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 19 — WORKFLOW

The Reliable Step-by-Step Workflow

91. Choose shared or static intentionally.
92. Configure CMake against the matching installed package.
93. Build Release or RelWithDebInfo.
94. Test the release from a clean directory.
95. Inspect runtime dependencies before shipping.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 19 — WORKING CODE

A Minimal Pattern You Can Build On

Example

CMAKE • COPY-READY

```cmake
find_package(SDL3 CONFIG REQUIRED COMPONENTS SDL3-static)
add_executable(app src/main.c)
target_link_libraries(app PRIVATE SDL3::SDL3-static)

# Only use this form when the installed SDL package actually
# provides SDL3::SDL3-static and all required static dependencies.
```

What to Notice

- Choose shared or static intentionally.
- Configure CMake against the matching installed package.
- Build Release or RelWithDebInfo.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 19 — FAILURE MODES

Mistakes That Waste the Most Time

1. SDL3::SDL3-static is not guaranteed unless the package provides it

SDL3::SDL3-static is not guaranteed unless the package provides it.

2. Static SDL may still depend on system libraries

Static SDL may still depend on system libraries.

3. A program that runs on the development machine may fail on a clean target because PATH/RPATH hid a missing library

A program that runs on the development machine may fail on a clean target because PATH/RPATH hid a missing library.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 19 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain shared linking keeps SDL in a separate runtime library..
- I can explain static linking copies library code into the final program..
- I can explain cMake target names are safer than hand-written -l flags..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Build one shared and one static variant if your platform package supports both. Compare output size and runtime dependency lists.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

CHAPTER 20 — AUDIO & SHIPPING

Cross-Platform Shipping: Windows, Linux, and macOS

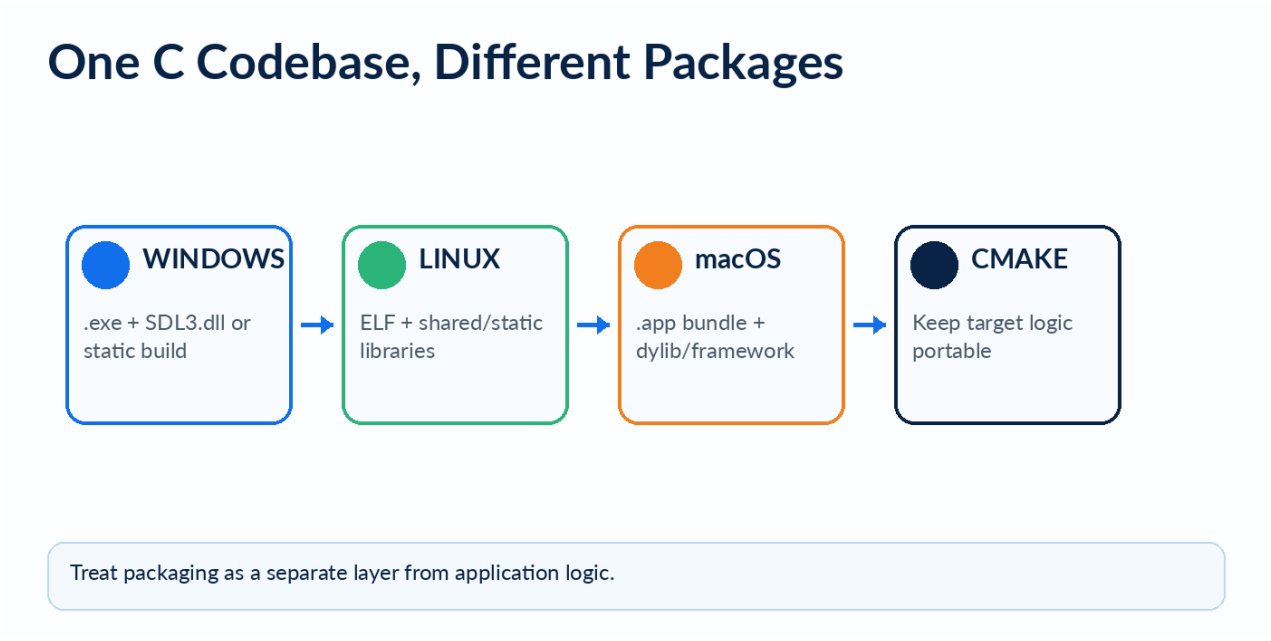
SDL3 removes a huge amount of platform-specific multimedia code, but packaging still belongs to each operating system. Keep platform conditionals at the project and packaging edge, not scattered through scene logic.

Core Mental Model

- Windows shared builds need SDL3.dll discoverable at runtime.
- Linux distributions vary; test on the oldest environment you promise to support.
- macOS applications normally ship as bundles and may require signing/notarization.
- Assets should use portable paths and be included in the release package.

Key APIs / Tools

- CMAKE_PREFIX_PATH
- CMAKE_TOOLCHAIN_FILE
- CMAKE_RUNTIME_OUTPUT_DIRECTORY
- $<TARGET_FILE:...>
- install(TARGETS ...)



Visual model for Chapter 20.

CHAPTER 20 — API MAP

The Small API Surface You Actually Need

The fastest way to learn SDL3 is to work from a small, intentional set of functions. Learn what each one owns or changes, then expand only when your project needs a capability.

`CMAKE_PREFIX_PATH` — is a central function, type, target, or configuration point for this chapter.

`CMAKE_TOOLCHAIN_FILE` — is a central function, type, target, or configuration point for this chapter.

`CMAKE_RUNTIME_OUTPUT_DIRECTORY` — is a central function, type, target, or configuration point for this chapter.

`$<TARGET_FILE:...>` — is a central function, type, target, or configuration point for this chapter.

`install(TARGETS ...)` — is a central function, type, target, or configuration point for this chapter.

Working Rule

Every SDL call belongs to one of four categories: create/own a resource, mutate state, query state, or perform work. If you classify an unfamiliar function this way, its role becomes much easier to remember.

Read Signatures SDL3 uses bool for many operations that were integer error codes in older examples. Always read the current signature before copying historical code.

CHAPTER 20 — WORKING CODE

A Minimal Pattern You Can Build On

Example

CMAKE • COPY-READY

```cmake
install(TARGETS app RUNTIME DESTINATION .)
install(DIRECTORY assets DESTINATION .)

# Configure: cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
# Build:     cmake --build build
# Stage:     cmake --install build --prefix stage
```

What to Notice

- Build Release on each target or trusted cross toolchain.
- Stage executable, runtime libraries, and assets.
- Run from a clean staging directory.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 20 — WORKFLOW

The Reliable Step-by-Step Workflow

96. Build Release on each target or trusted cross toolchain.
97. Stage executable, runtime libraries, and assets.
98. Run from a clean staging directory.
99. Test DPI, input, audio, and fullscreen/window behavior.
100. Archive the exact artifacts you tested.

Why This Order Works

The sequence protects ownership and makes failures local. Each step either establishes a prerequisite for the next one or transforms data into a representation that the next stage expects. When something fails, the last successful step tells you exactly where to investigate.

Think in State Transitions

Graphics code is often easier when you describe it as transitions: not initialized -> initialized, no resource -> valid resource, queued input -> application action, model state -> rendered pixels. A transition should have a clear success condition and a clear cleanup path.

Performance Optimize structure before micro-optimizing calls. Loading once, reusing resources, avoiding needless allocation, and separating update from render usually matters more than clever syntax.

CHAPTER 20 — WORKING CODE

A Minimal Pattern You Can Build On

Example

CMAKE • COPY-READY

```cmake
if(WIN32 AND TARGET SDL3::SDL3-shared)
  add_custom_command(TARGET app POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy
      $<TARGET_FILE:SDL3::SDL3-shared>
      $<TARGET_FILE_DIR:app>)
endif()
```

What to Notice

- Build Release on each target or trusted cross toolchain.
- Stage executable, runtime libraries, and assets.
- Run from a clean staging directory.

Do not treat this code as a magic recipe. Type it, run it, then deliberately change one assumption. Change a size, color, flag, timing value, or asset path and observe the consequence. That is how the API becomes a tool rather than a list of names.

Compile Discipline Keep warnings enabled. A graphics program can look correct while still containing lifetime, conversion, or uninitialized-state bugs.

CHAPTER 20 — FAILURE MODES

Mistakes That Waste the Most Time

1. Packaging is not just “copy the exe

Packaging is not just “copy the exe.”

2. Case-sensitive asset paths can fail on Linux after working on Windows

Case-sensitive asset paths can fail on Linux after working on Windows.

3. macOS and Windows security warnings can appear if release signing is ignored

macOS and Windows security warnings can appear if release signing is ignored.

A Better Debugging Question

Instead of asking “Why does SDL not work?”, ask a narrow question: Is the resource valid? Is the current render target the one I think it is? Are the coordinates in the same space as the renderer? Did I clear after drawing? Did the event pump run? Is the asset path resolved from the process working directory? Narrow questions produce narrow fixes.

Debug Order Validate return value -> print SDL_GetError -> reduce the scene -> verify lifetime -> verify coordinates/state -> reintroduce complexity.

CHAPTER 20 — PRACTICE

Checklist and Deliberate Practice

Before You Leave This Chapter

- I can explain windows shared builds need SDL3.dll discoverable at runtime..
- I can explain linux distributions vary; test on the oldest environment you promise to support..
- I can explain macOS applications normally ship as bundles and may require signing/notarization..
- I can identify which objects I must destroy.
- I can reproduce the example without copying it line for line.

Build This

Create a release-staging CMake target that collects the executable and an assets directory. On Windows shared builds, also copy SDL3.dll with a generator expression.

Stretch Goal

Add one visible diagnostic overlay that proves the relevant state is changing: coordinates, frame number, input state, resource count, audio trigger count, or current target. The overlay turns invisible logic into evidence you can inspect.

Definition of Done The program must build in Debug and Release, exit cleanly, report no sanitizer error in your sanitized profile, and still work after you move it to a fresh build directory.

APPENDIX A

SDL3 Function Quick Reference

Initialization

SDL_Init, SDL_Quit, SDL_GetError, SDL_Log

Window

SDL_CreateWindow, SDL_CreateWindowAndRenderer, SDL_DestroyWindow

Renderer

SDL_CreateRenderer, SDL_SetRenderDrawColor, SDL_RenderClear, SDL_RenderPresent

Primitives

SDL_RenderPoint(s), SDL_RenderLine(s), SDL_RenderRect(s), SDL_RenderFillRect(s)

Textures

SDL_CreateTexture, SDL_CreateTextureFromSurface, SDL_RenderTexture, SDL_DestroyTexture

Targets

SDL_SetRenderTarget, SDL_GetRenderTarget

Scaling

SDL_SetRenderLogicalPresentation, SDL_SetTextureScaleMode

Events

SDL_PollEvent, SDL_Event, SDL_EventType

Keyboard/Mouse

SDL_GetKeyboardState, SDL_GetMouseState, SDL_GetRelativeMouseState

Gamepad

SDL_GetGamepads, SDL_OpenGamepad, SDL_GetGamepadButton, SDL_GetGamepadAxis

Timing

SDL_GetTicks, SDL_GetTicksNS, SDL_DelayNS

Audio

SDL_OpenAudioDeviceStream, SDL_PutAudioStreamData, SDL_ResumeAudioStreamDevice, SDL_DestroyAudioStream

APPENDIX A — QUICK REFERENCE

Renderer Geometry and State

`SDL_RenderPoint(renderer, x, y)`
Draw one point using float coordinates.

`SDL_RenderLine(renderer, x1, y1, x2, y2)`
Draw one line.

`SDL_RenderRect(renderer, &rect)`
Draw rectangle outline.

`SDL_RenderFillRect(renderer, &rect)`
Draw filled rectangle.

`SDL_SetRenderDrawColor(renderer, r,g,b,a)`
Set primitive and clear color.

> **Remember** Current signatures and thread-safety notes belong to the official SDL Wiki. Treat this appendix as a navigation map, not a replacement for reference documentation.

APPENDIX A — QUICK REFERENCE

Texture and Target Operations

`SDL_CreateTextureFromSurface`
Create renderer texture from CPU surface.

`SDL_RenderTexture`
Render full or partial texture with SDL_FRect rectangles.

`SDL_SetTextureScaleMode`
Choose nearest/linear/pixel-art scaling.

`SDL_SetRenderTarget`
Switch drawing to target texture or NULL window.

`SDL_DestroyTexture`
Release texture before destroying renderer.

> **Remember** Current signatures and thread-safety notes belong to the official SDL Wiki. Treat this appendix as a navigation map, not a replacement for reference documentation.

APPENDIX A — QUICK REFERENCE

Input and Event Operations

`SDL_PollEvent`
Drain discrete events.

`SDL_GetKeyboardState`
Read held keys by scancode.

`SDL_GetMouseState`
Read cached window-relative mouse state.

`SDL_GetGamepads`
Enumerate connected gamepad IDs.

`SDL_OpenGamepad`
Open standardized gamepad interface.

Remember Current signatures and thread-safety notes belong to the official SDL Wiki. Treat this appendix as a navigation map, not a replacement for reference documentation.

APPENDIX A — QUICK REFERENCE

Audio and Add-on Libraries

`SDL_OpenAudioDeviceStream`
Convenient device+stream opening.

`SDL_PutAudioStreamData`
Queue application audio data.

`IMG_LoadTexture`
SDL_image: load file directly to texture.

`TTF_CreateText`
SDL_ttf: create reusable text object.

`TTF_DrawRendererText`
Draw text object through its renderer text engine.

Remember Current signatures and thread-safety notes belong to the official SDL Wiki. Treat this appendix as a navigation map, not a replacement for reference documentation.

APPENDIX B

CMake and Build Recipes

Portable SDL3 Target

CMAKE • COPY-READY

```cmake
cmake_minimum_required(VERSION 3.25)
project(SDL3BookDemo LANGUAGES C)

set(CMAKE_C_STANDARD 23)
set(CMAKE_C_STANDARD_REQUIRED ON)
set(CMAKE_C_EXTENSIONS OFF)

find_package(SDL3 CONFIG REQUIRED)
add_executable(SDL3BookDemo src/main.c)
target_link_libraries(SDL3BookDemo PRIVATE SDL3::SDL3)

target_compile_options(SDL3BookDemo PRIVATE
    $<$<C_COMPILER_ID:GNU>:-Wall -Wextra -Wpedantic>)
```

Fedora 44

TERMINAL • COPY-READY

```
sudo dnf install SDL3-devel
pkg-config --modversion sdl3
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Debug
cmake --build build
```

Release

TERMINAL • COPY-READY

```
cmake -S . -B build-release -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build-release
```

APPENDIX B — STATIC AND SHARED

Choose the Library Form Explicitly When Available

Shared

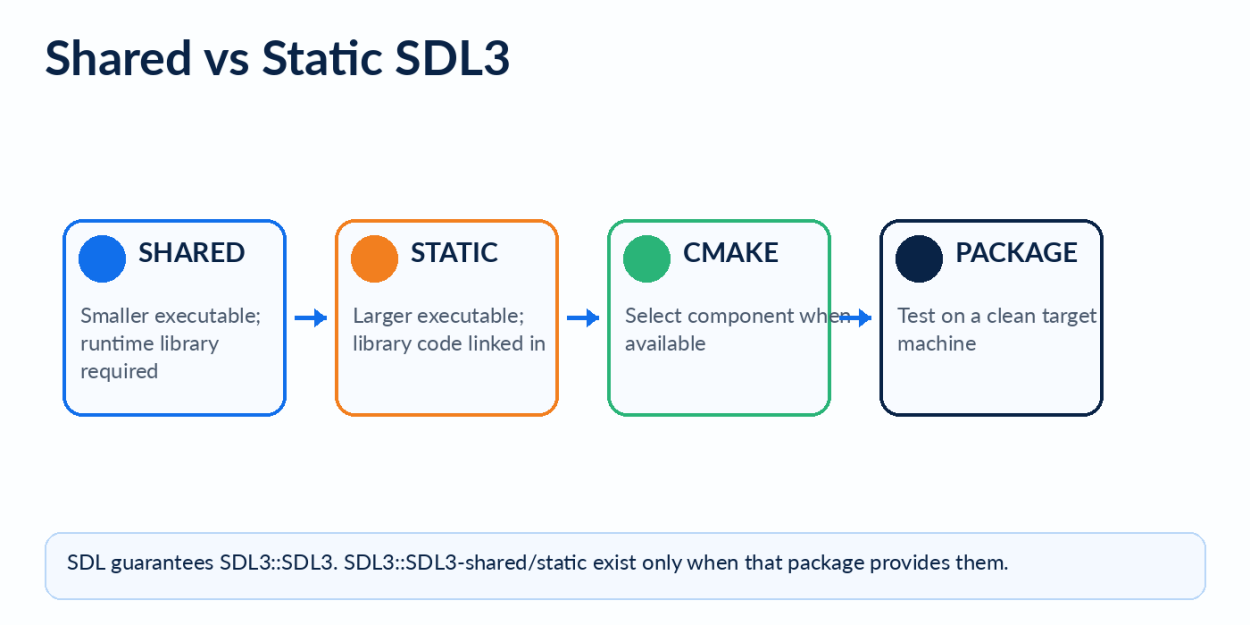
find_package(SDL3 CONFIG REQUIRED COMPONENTS SDL3-shared)

Use SDL3::SDL3 or SDL3::SDL3-shared when provided. Ship the runtime shared library where the platform expects it.

Static

find_package(SDL3 CONFIG REQUIRED COMPONENTS SDL3-static)

Use SDL3::SDL3-static only when the installed package exposes it and static dependencies are available.



CMake makes the linking choice explicit and reviewable.

Guarantee SDL documents that SDL3::SDL3 is guaranteed. The explicit shared/static imported targets are not guaranteed to exist in every installed package.

APPENDIX B — CLION

A Clean CLion Profile Set

- Debug: GCC 16.1, Ninja, -DCMAKE_BUILD_TYPE=Debug.
- Release: same toolchain, -DCMAKE_BUILD_TYPE=Release.
- Sanitized: Debug plus -fsanitize=address,undefined compile and link options.
- Do not use different compilers accidentally between terminal and IDE.
- If CMake finds the wrong SDL, inspect CMAKE_PREFIX_PATH and remove stale CMake cache entries.

CLion 2026.2 Toolchains define compilers/build tools/debugger. CMake profiles select a toolchain and build configuration for the project.

APPENDIX C

Common Error Patterns and Fixes

CMake cannot find SDL3

Likely cause: SDL3-devel is missing, installed outside CMake search prefixes, or a stale cache points elsewhere.

Fix: Verify pkg-config, inspect CMAKE_PREFIX_PATH, and delete/reload the build directory.

Header not found

Likely cause: The project is bypassing imported targets or using an incorrect include path.

Fix: Link SDL3::SDL3 and include <SDL3/SDL.h>.

Diagnostic Habit Always log the exact operation that failed, not just SDL_GetError by itself. Context makes logs actionable.

APPENDIX C

Common Error Patterns and Fixes

Window opens then freezes

Likely cause: The application is not pumping events.

Fix: Run SDL_PollEvent each frame.

Black window

Likely cause: You may be clearing but not drawing, drawing to the wrong target, or forgetting SDL_RenderPresent.

Fix: Verify target, clear color, one primitive, then present.

Diagnostic Habit Always log the exact operation that failed, not just SDL_GetError by itself. Context makes logs actionable.

APPENDIX C

Common Error Patterns and Fixes

Image does not load

Likely cause: Working directory or path case is wrong.

Fix: Log the asset path and SDL_GetError; centralize asset root.

Pixel art is blurry

Likely cause: Linear scale mode or non-integer scaling.

Fix: Use logical integer scaling and nearest/pixel-art texture mode.

Diagnostic Habit Always log the exact operation that failed, not just SDL_GetError by itself. Context makes logs actionable.

APPENDIX C

Common Error Patterns and Fixes

No audio

Likely cause: Stream is paused, no data queued, or format is wrong.

Fix: Queue valid PCM, resume stream device, log errors.

Release fails on another machine

Likely cause: Missing shared library or packaging dependency.

Fix: Test from a clean staging directory/VM and inspect runtime dependencies.

Diagnostic Habit Always log the exact operation that failed, not just SDL_GetError by itself. Context makes logs actionable.

APPENDIX D — PRACTICE PROJECTS

Projects That Turn API Knowledge into Skill

1. Primitive Plotter

Grid, axes, points, polylines, rectangles, zoom and pan.

Skills: Primitives, coordinates, mouse input.

2. Sprite Inspector

Load a texture, draw source rectangles, change scale mode, preview animation clips.

Skills: Textures, sprite sheets, timing.

Constraint Keep each project under roughly 500 lines of C until the feature is understood. Add architecture only after repetition reveals a real pattern.

APPENDIX D — PRACTICE PROJECTS

Projects That Turn API Knowledge into Skill

3. Input Laboratory

Visualize keyboard, mouse, touch, and gamepad state in real time.

Skills: Events, state queries, device hotplug.

4. Audio Trigger Board

Buttons trigger short WAV effects with per-sound volume and timing logs.

Skills: Audio streams, UI, latency.

Constraint Keep each project under roughly 500 lines of C until the feature is understood. Add architecture only after repetition reveals a real pattern.

APPENDIX D — PRACTICE PROJECTS

Projects That Turn API Knowledge into Skill

5. Mini Scene Composer

Background layer, sprites, render target, HUD, camera rectangle.

Skills: Layers, targets, logical presentation.

6. Cross-Platform Demo

Package one tiny demo for Linux, Windows, and macOS.

Skills: CMake, shared/static linking, staging.

Constraint Keep each project under roughly 500 lines of C until the feature is understood. Add architecture only after repetition reveals a real pattern.

APPENDIX E

Official References and Further Reading

SDL3 Wiki

Primary API reference and migration material.

https://wiki.libsdl.org/SDL3/FrontPage

SDL3 CMake Guide

Official imported-target, shared/static, vendoring, and install guidance.

https://wiki.libsdl.org/SDL3/README-cmake

SDL3 Migration Guide

Essential when comparing older SDL2 material with SDL3.

https://wiki.libsdl.org/SDL3/README-migration

SDL_image 3

Image format loading add-on.

https://wiki.libsdl.org/SDL3_image

SDL_ttf 3

TrueType/OpenType text add-on based on FreeType/HarfBuzz.

https://wiki.libsdl.org/SDL3_ttf/FrontPage

CLion 2026.2 Help

Toolchains, CMake profiles, debugger, and project configuration.

https://www.jetbrains.com/help/clion/

Fedora SDL3 Packages

Fedora package versions and development package information.

https://packages.fedoraproject.org/pkgs/SDL3/SDL3-devel/

CMake Documentation

Authoritative CMake language and build-system reference.

https://cmake.org/cmake/help/latest/

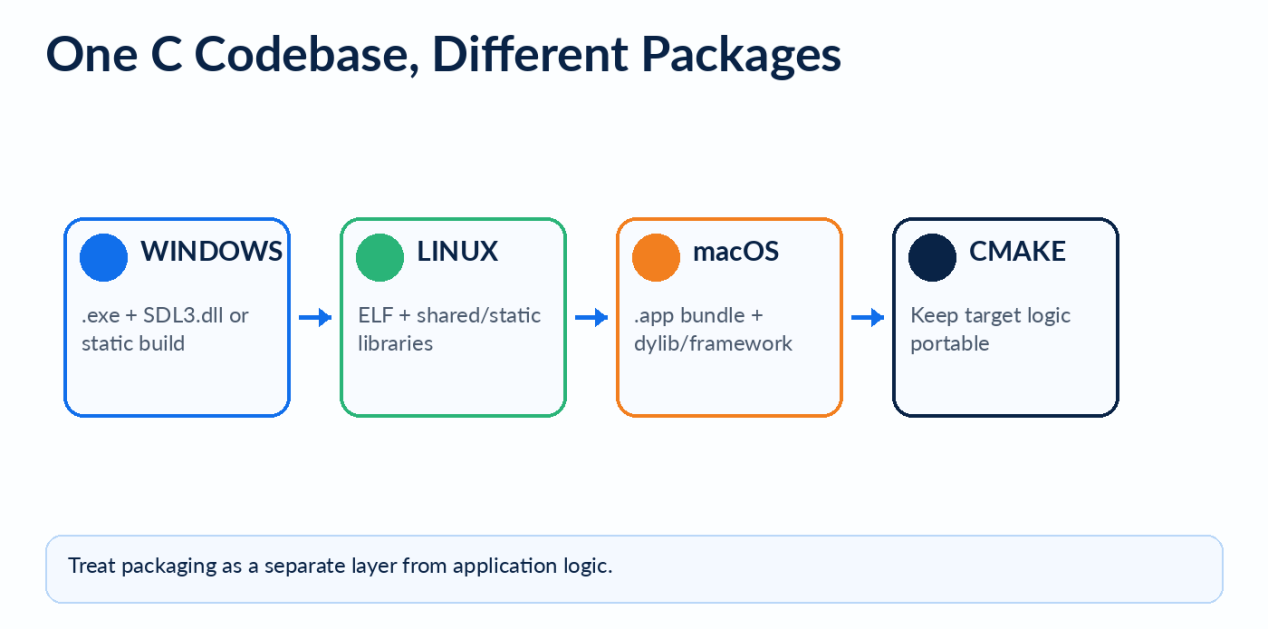
CLOSING NOTE

The Best Next Step Is to Draw Something

SDL3 becomes understandable when it is used. Open a window. Draw one line. Move it with the keyboard. Load one texture. Add a frame timer. Add a sound. Then package the program and run it on another machine. Each step is small, but together they teach the whole path from C source to visible, interactive software.

The library is powerful precisely because it does not require you to adopt a large engine or a complicated application framework. It gives you cross-platform access to the multimedia building blocks and lets you decide how much structure your program actually needs.

Final Principle Keep the code direct, keep ownership explicit, keep the frame loop understandable, and measure before optimizing. Clarity and performance are allies when the design is disciplined.



One codebase can travel — if build and packaging decisions remain deliberate.

APPENDIX F

Guided Labs — Build, Observe, Measure, Extend

These labs turn the book into a working graphics-programming course. Each chapter receives two practical pages: first build a focused experiment with copy-ready source, then deliberately change conditions and observe what SDL3 actually does. The goal is not merely to finish examples, but to develop the habit of measuring state, ownership, timing, and visual output.

GUIDED LAB 01 — BUILD

Toolchain Confidence Lab

GOAL	STARTING POINT
Prove that GCC 16.1, CMake, CLion, and SDL3 are all using the same installation and build graph.	Start from an empty folder and a working SDL3 development package.

Build It

1. Create a minimal CMake project that declares C23 explicitly.
2. Configure it from a clean build directory.
3. Build from both CLion and the terminal.
4. Print the SDL version and renderer name at runtime.
5. Delete the build directory and prove the project configures again.

Copy-Ready Focus Listing

CMAKE • COPY-READY

```cmake
cmake_minimum_required(VERSION 3.25)
project(SDL3Lab LANGUAGES C)
set(CMAKE_C_STANDARD 23)
set(CMAKE_C_STANDARD_REQUIRED ON)
set(CMAKE_C_EXTENSIONS OFF)
find_package(SDL3 CONFIG REQUIRED)
add_executable(SDL3Lab src/main.c)
target_link_libraries(SDL3Lab PRIVATE SDL3::SDL3)
target_compile_options(SDL3Lab PRIVATE
    $<$<C_COMPILER_ID:GNU>:-Wall -Wextra -Wpedantic>)
```

Expected Result A clean configure/build proves the IDE and terminal are describing the same target rather than two unrelated environments.

Try These Variations

- Switch the build type between Debug and Release and compare compiler flags.
- Delete the build directory and rebuild from a completely clean configure step.
- Run the executable once from CLion and once directly from the terminal.
- Temporarily break SDL discovery, read the CMake error, then restore the package path.

IDE Verification

Rebuild with warnings enabled, run from a fresh working directory, and confirm that the same source behaves correctly outside the IDE. Save one screenshot or console log as evidence.

Ownership Check

Name every resource or external dependency this lab relies on and identify exactly who creates it, who uses it, and what action releases or replaces it.

GUIDED LAB 01 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Change the compiler path in CLion and observe CMake reconfigure.
- Turn one warning into an error and verify the build fails for the expected reason.
- Move the project directory and prove no absolute include paths were required.
- Inspect the CMake cache and identify where SDL3 was found.

What to Watch For

- Compiler mismatch between terminal and IDE.
- A stale CMake cache hiding a changed toolchain.
- Hard-coded include or library paths.
- Runtime library lookup that differs from compile-time discovery.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Treat the build system as part of the program. Reproducibility is a performance feature because it shortens every debug and profiling cycle.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the toolchain confidence exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 02 — BUILD

Window Lifetime Lab

GOAL	STARTING POINT
Build a resizable high-pixel-density window and make every lifetime transition visible in the log.	Use the Chapter 1 CMake target and add one source file.

Build It

1. Initialize only SDL_INIT_VIDEO.
2. Create a resizable high-pixel-density window.
3. Handle quit and close-request events.
4. Log resize events with their new dimensions.
5. Destroy the window before SDL_Quit.

Copy-Ready Focus Listing

C23 • COPY-READY

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

int main(void) {
    if (!SDL_Init(SDL_INIT_VIDEO)) return 1;
    SDL_Window *w = SDL_CreateWindow(
        "Window Lab", 960, 540,
        SDL_WINDOW_RESIZABLE | SDL_WINDOW_HIGH_PIXEL_DENSITY);
```

```
    if (!w) { SDL_Quit(); return 1; }
    bool running = true;
    SDL_Event e;
    while (running) {
        while (SDL_PollEvent(&e)) {
            if (e.type == SDL_EVENT_QUIT ||
                e.type == SDL_EVENT_WINDOW_CLOSE_REQUESTED)
                running = false;
            if (e.type == SDL_EVENT_WINDOW_RESIZED)
                SDL_Log("size=%dx%d", e.window.data1, e.window.data2);
        }
    }
    SDL_DestroyWindow(w);
    SDL_Quit();
    return 0;
}
```

Expected Result The window remains responsive while you resize it, and every size change appears in the log.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Toggle SDL_WINDOW_RESIZABLE and compare emitted window events.
- Move the window between normal- and high-DPI displays if available.
- Change the initial width and height and confirm the lifetime logic is unchanged.
- Log focus, minimize, restore, and close-request events in addition to resize.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 02 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Remove event polling and observe how the OS treats the window.
- Toggle SDL_WINDOW_RESIZABLE and compare behavior.
- Create a second window and track each window ID.
- Add Escape-to-quit and compare event-driven exit with close-request exit.

What to Watch For

- Continuing after SDL_CreateWindow returns NULL.
- Destroying resources in the wrong order.
- Assuming logical size equals pixel size on high-DPI displays.
- Doing heavy work before the event queue is pumped.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.

State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle A window is not merely a rectangle; it is an OS-managed resource with a lifecycle. Make that lifecycle explicit before adding graphics.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the window lifetime exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 03 — BUILD

Renderer State Lab

GOAL	STARTING POINT
Learn which renderer settings persist and how the current draw color affects subsequent operations.	Create a window and renderer before entering the loop.

Build It

1. Clear to a dark background.
2. Draw three colored rectangles.
3. Change only the draw color between calls.
4. Present once per frame.
5. Log SDL_GetRendererName once at startup.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
const char *name = SDL_GetRendererName(renderer);
SDL_Log("renderer=%s", name ? name : "unknown");

SDL_SetRenderDrawColor(renderer, 12, 18, 30, 255);
SDL_RenderClear(renderer);

SDL_FRect a = {40, 40, 120, 80};
SDL_FRect b = {180, 40, 120, 80};
SDL_FRect c = {320, 40, 120, 80};
SDL_SetRenderDrawColor(renderer, 244, 63, 94, 255);
SDL_RenderFillRect(renderer, &a);
SDL_SetRenderDrawColor(renderer, 34, 197, 94, 255);
SDL_RenderFillRect(renderer, &b);
SDL_SetRenderDrawColor(renderer, 59, 130, 246, 255);
SDL_RenderFillRect(renderer, &c);
SDL_RenderPresent(renderer);
```

Expected Result Three colored rectangles appear over one dark frame, demonstrating that draw color is mutable renderer state.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Change the clear color once per second so renderer state becomes visible.
- Draw the same rectangle before and after a color change and compare the result.
- Query and log the renderer name, then keep the rendering code driver-neutral.
- Move one drawing call before SDL_RenderClear and explain why it disappears.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 03 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Move SDL_RenderClear after drawing and note what disappears.
- Call SDL_RenderPresent twice without new drawing.
- Use float draw colors and compare the API style.
- Change the renderer background every second to visualize state transitions.

What to Watch For

- Clearing after drawing.
- Presenting before the frame is complete.
- Assuming draw color belongs to an individual shape.
- Ignoring false return values from draw operations.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Renderer state is simple but global to the rendering context. Group operations by shared state to make intent obvious and reduce needless changes.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?

- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the renderer state exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 04 — BUILD

Frame Loop Lab

GOAL	STARTING POINT
Separate event handling, update, and render so each stage can be reasoned about independently.	Begin with a working window and renderer.

Build It

1. Poll all pending events.
2. Compute elapsed time.
3. Update one moving value.
4. Render from the updated state.
5. Present exactly once.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
static Uint64 previous = 0;
Uint64 now = SDL_GetTicksNS();
if (previous == 0) previous = now;
float dt = (float)(now - previous) / 1000000000.0f;
previous = now;
if (dt > 0.05f) dt = 0.05f;

while (SDL_PollEvent(&event)) {
    if (event.type == SDL_EVENT_QUIT) running = false;
}

x += velocity * dt;
SDL_SetRenderDrawColor(renderer, 10, 15, 28, 255);
SDL_RenderClear(renderer);
SDL_FRect box = {x, 180.0f, 48.0f, 48.0f};
SDL_SetRenderDrawColor(renderer, 80, 160, 255, 255);
SDL_RenderFillRect(renderer, &box);
SDL_RenderPresent(renderer);
```

Expected Result A rectangle moves at nearly the same perceived speed even when frame duration varies.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Cap a moving value at two different speeds while keeping update order unchanged.
- Inject a deliberate 100 ms pause and observe the effect on elapsed time.
- Count updates and presents separately to verify one present per rendered frame.
- Move event polling after rendering once, observe responsiveness, then restore the correct order.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 04 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Artificially sleep for 10 ms and compare movement.
- Remove the dt clamp, then drag the window to create a long frame.
- Log update time separately from render time.
- Split the loop into HandleEvents, Update, and Render functions.

What to Watch For

- Mixing input mutation with drawing code.
- Using frame count instead of elapsed time for motion.
- Allowing one stalled frame to create a huge simulation jump.
- Presenting more than once per logical frame.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle A clean loop is the architecture of a small graphics program. Once responsibilities are separated, profiling and optimization become much easier.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the frame loop exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 05 — BUILD

Coordinate Conversion Lab

GOAL	STARTING POINT
Make screen position, logical rendering coordinates, and color values visible instead of guessing them.	Enable logical presentation before the event loop.

Build It

1. Set a fixed logical resolution.
2. Convert mouse events into render coordinates.
3. Draw a marker at the converted location.
4. Display or log the coordinates.
5. Try letterbox and integer-scale modes.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
SDL_SetRenderLogicalPresentation(
    renderer, 640, 360,
    SDL_LOGICAL_PRESENTATION_LETTERBOX);

while (SDL_PollEvent(&event)) {
    if (event.type == SDL_EVENT_MOUSE_MOTION) {
        SDL_ConvertEventToRenderCoordinates(renderer, &event);
        marker_x = event.motion.x;
        marker_y = event.motion.y;
    }
}

SDL_FRect marker = {marker_x - 4.0f, marker_y - 4.0f, 8.0f, 8.0f};
SDL_SetRenderDrawColor(renderer, 255, 200, 40, 255);
SDL_RenderFillRect(renderer, &marker);
```

Expected Result The marker follows the cursor correctly even when the window is resized or letterboxed.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Resize the window to wide, tall, and square aspect ratios and compare mapped coordinates.
- Switch between letterbox and integer-scale logical presentation modes.
- Draw both raw mouse coordinates and converted render coordinates at the same time.
- Click the four logical corners and verify the converted values are close to expected edges.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 05 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Compare raw event coordinates with converted render coordinates.
- Switch to SDL_LOGICAL_PRESENTATION_INTEGER_SCALE.
- Draw the logical center and verify it stays centered.

- Use alpha values of 64, 128, and 255 over the same background.

What to Watch For

- Mixing window pixels with logical coordinates.
- Forgetting event coordinate conversion when logical presentation is active.
- Assuming Y grows upward.
- Using alpha without considering blend mode.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Coordinate bugs masquerade as rendering bugs. Decide which coordinate space each value belongs to and name variables accordingly.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the coordinate conversion exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 06 — BUILD

Primitive Composition Lab

GOAL	STARTING POINT
Build a recognizable scene using only points, lines, and rectangles before introducing assets.	Work inside a normal render loop.

Build It

1. Draw a horizon line.
2. Build a house from rectangles.
3. Add a roof outline with lines.
4. Place stars using points.
5. Change colors in logical groups.

Copy-Ready Focus Listing

C23 • COPY-READY

```
SDL_SetRenderDrawColor(renderer, 18, 26, 46, 255);
SDL_RenderClear(renderer);
SDL_SetRenderDrawColor(renderer, 90, 180, 255, 255);
SDL_RenderLine(renderer, 0, 250, 640, 250);

SDL_FRect house = {230, 150, 180, 100};
SDL_SetRenderDrawColor(renderer, 220, 120, 70, 255);
SDL_RenderFillRect(renderer, &house);

SDL_SetRenderDrawColor(renderer, 245, 220, 150, 255);
SDL_RenderLine(renderer, 220, 150, 320, 80);
SDL_RenderLine(renderer, 320, 80, 420, 150);

SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
for (int i = 0; i < 20; ++i)
    SDL_RenderPoint(renderer, (float)(i * 31 % 640), (float)(i * 47 % 120));
SDL_RenderPresent(renderer);
```

Expected Result A simple house and sky appear with no image assets, proving how far primitive composition can go.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Reorder the same primitives and document which shapes cover others.
- Build one object entirely from rectangles and lines, then move it as a group.
- Change alpha values and observe how blending affects overlapping shapes.
- Create a simple grid helper and use it to reason about placement numerically.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 06 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Change draw order so the roof is hidden by the house.
- Animate one star horizontally.
- Create a reusable DrawHouse function.
- Group shared-color calls and compare readability.

What to Watch For

- Off-by-one coordinates at edges.
- Using integer rectangles where subpixel motion is desired.
- Changing color before every single point unnecessarily.
- Forgetting the final present call.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.

State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Primitive drawing is ideal for debug overlays, editors, prototypes, and HUD geometry even in asset-heavy projects.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the primitive composition exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 07 — BUILD

Image Pipeline Lab

GOAL	STARTING POINT
Load an image into a CPU surface, create a renderer texture, release the surface, and draw the texture repeatedly.	Place a BMP/PNG/JPEG file in an assets directory.

Build It

1. Load the file once at startup.
2. Check the returned surface.
3. Create a texture from the surface.
4. Destroy the surface immediately after texture creation.
5. Reuse the texture every frame and destroy it at shutdown.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
SDL_Surface *surface = SDL_LoadSurface("assets/picture.png");
if (!surface) {
    SDL_Log("load: %s", SDL_GetError());
    return 1;
}
SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, surface);
SDL_DestroySurface(surface);
if (!texture) {
    SDL_Log("texture: %s", SDL_GetError());
    return 1;
}

SDL_FRect dst = {80.0f, 50.0f, (float)texture->w, (float)texture->h};
SDL_RenderTexture(renderer, texture, NULL, &dst);

/* at shutdown */
SDL_DestroyTexture(texture);
```

Expected Result The image appears each frame while only the GPU-friendly texture remains alive during rendering.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Load the same asset twice, then refactor so one texture is reused.
- Intentionally use a wrong asset path and verify the exact failure is logged.
- Render one texture at multiple destination sizes without reloading it.
- Destroy the source surface immediately after texture creation and confirm rendering still works.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 07 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Scale dst to half and double size.
- Load two textures and swap them with a key press.
- Deliberately use a bad path and verify the error message is specific.
- Time repeated loading inside the loop to see why it is a bad design.

What to Watch For

- Leaking the surface after texture creation.
- Loading files every frame.
- Using a relative path without understanding the working directory.
- Continuing after a failed texture creation.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Resource conversion is a boundary: CPU-side asset preparation should usually happen outside the hot render loop.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the image pipeline exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 08 — BUILD

Sprite Timing Lab

GOAL	STARTING POINT
Advance sprite frames with elapsed time rather than frame count.	Assume a texture contains equally sized frames in a horizontal strip.

Build It

1. Accumulate dt.
2. Advance when the accumulator reaches frame_time.
3. Wrap the frame index.
4. Build an SDL_FRect source rectangle.
5. Render into a destination rectangle.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
animation_time += dt;
const float frame_time = 0.10f;
while (animation_time >= frame_time) {
    animation_time -= frame_time;
    frame = (frame + 1) % frame_count;
}

SDL_FRect src = {
    (float)(frame * frame_w), 0.0f,
    (float)frame_w, (float)frame_h
};
SDL_FRect dst = {x, y, 64.0f, 64.0f};
SDL_RenderTexture(renderer, sprite_sheet, &src, &dst);
```

Expected Result The animation speed remains stable even if the application occasionally renders a slower frame.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Run the animation at 0.05 s, 0.10 s, and 0.20 s per frame and compare perceived speed.
- Create a non-looping one-shot clip and stop on its final frame.
- Change destination position independently of animation frame selection.
- Pause animation while movement continues to prove the two states are separate.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 08 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Use 0.05, 0.10, and 0.20 second frame times.
- Make walk loop while jump stops on its final frame.
- Separate movement state from animation state.
- Log frame index and dt during a resize stall.

What to Watch For

- Advancing exactly one frame per render.
- Resetting the accumulator instead of subtracting frame_time.
- Mixing source pixel dimensions with destination world size.
- Letting animation choose game logic instead of reflecting it.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Animation is time-domain state. Keep its clock explicit so you can pause, slow, loop, or replace clips without rewriting movement logic.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the sprite timing exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 09 — BUILD

HUD Text Lab

GOAL	STARTING POINT
Render text with SDL_ttf 3, turn the resulting surface into a texture, and update only when the text changes.	Link SDL3_ttf and have a redistributable font available.

Build It

1. Initialize SDL_ttf.
2. Open one font.
3. Render a string to a surface.
4. Convert the surface to a texture.
5. Cache the texture until the string changes.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
#include <SDL3_ttf/SDL_ttf.h>

TTF_Init();
TTF_Font *font = TTF_OpenFont("assets/ui.ttf", 24.0f);
SDL_Color white = {255, 255, 255, 255};
SDL_Surface *s = TTF_RenderText_Blended(
    font, "SCORE 1200", 0, white);
SDL_Texture *text = SDL_CreateTextureFromSurface(renderer, s);
SDL_DestroySurface(s);

SDL_FRect dst = {20, 18, (float)text->w, (float)text->h};
SDL_RenderTexture(renderer, text, NULL, &dst);

SDL_DestroyTexture(text);
TTF_CloseFont(font);
TTF_Quit();
```

Expected Result A crisp HUD label is rendered as an ordinary texture that can be positioned like any other graphical element.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Update the score every frame, then cache it and update only when the value changes.
- Render the same text at two font sizes and compare readability at the target resolution.
- Change foreground color against light and dark backgrounds to test contrast.
- Move HUD text between corners while keeping gameplay coordinates unchanged.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 09 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Regenerate the texture only when score changes.

- Compare 18, 24, and 36 point sizes.
- Measure contrast against light and dark backgrounds.
- Try a long UTF-8 string and verify the chosen font supports its glyphs.

What to Watch For

- Creating text textures every frame when content is unchanged.
- Forgetting to destroy intermediate surfaces.
- Shipping a font without checking its license.
- Assuming point size equals physical pixels on every display.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Text is often more expensive than rectangles. Cache stable labels and treat frequently changing counters as a deliberate update path.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the hud text exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 10 — BUILD

Render Target Lab

GOAL	STARTING POINT
Compose several scene layers into an off-screen target and then draw the composed texture to the window.	Create a renderer that supports target textures.

Build It

1. Create a texture with SDL_TEXTUREACCESS_TARGET.
2. Set it as the current render target.
3. Draw the background and sprites.
4. Restore the target to NULL.
5. Draw the composed texture and present.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
SDL_Texture *scene = SDL_CreateTexture(
    renderer, SDL_PIXELFORMAT_RGBA8888,
    SDL_TEXTUREACCESS_TARGET, 640, 360);

SDL_SetRenderTarget(renderer, scene);
SDL_SetRenderDrawColor(renderer, 18, 28, 52, 255);
SDL_RenderClear(renderer);
DrawBackground(renderer);
DrawSprites(renderer);

SDL_SetRenderTarget(renderer, NULL);
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderClear(renderer);
SDL_RenderTexture(renderer, scene, NULL, NULL);
SDL_RenderPresent(renderer);

SDL_DestroyTexture(scene);
```

Expected Result The scene is first built off-screen and then presented as one texture, ready for scaling or post-processing.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Render only the background into a target texture and draw dynamic sprites directly.
- Deliberately forget to restore the render target, observe the failure, then fix it.
- Scale the composed target when drawing it back to the window.
- Create a small secondary target suitable for a mini-map or preview panel.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 10 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Forget to restore NULL once and observe the symptom.
- Render the target at half resolution and scale it up.
- Draw UI after restoring the window target so it stays sharp.
- Cache a static background in its own target texture.

What to Watch For

- Using a texture without target access.
- Applying logical presentation twice.
- Leaving the wrong target active between passes.
- Destroying a target while it is still referenced by code.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.

Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Render targets turn composition into a pipeline. Explicit passes are easier to profile than one giant list of draw calls.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the render target exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 11 — BUILD

Pixel-Perfect Scaling Lab

GOAL	STARTING POINT
Keep low-resolution art crisp while the window changes size and pixel density.	Choose a logical resolution such as 320x180.

Build It

1. Enable high-pixel-density window creation.
2. Set integer-scale logical presentation.
3. Use nearest scale mode for pixel-art textures.
4. Resize through several aspect ratios.
5. Verify input coordinates still map correctly.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
SDL_SetRenderLogicalPresentation(
    renderer, 320, 180,
    SDL_LOGICAL_PRESENTATION_INTEGER_SCALE);

SDL_SetTextureScaleMode(sprite,
    SDL_SCALEMODE_NEAREST);

SDL_FRect dst = {100.0f, 80.0f, 16.0f, 16.0f};
SDL_RenderTexture(renderer, sprite, NULL, &dst);
```

Expected Result Pixel edges remain hard at integer magnifications and the logical scene does not change its coordinate system.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.

- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Compare nearest and linear texture scale modes on the same pixel-art texture.
- Resize through integer and non-integer scale ratios and capture the visible difference.
- Test logical presentation on at least three window aspect ratios.
- Verify mouse picking still targets the same logical location after resizing.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 11 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Switch to letterbox and compare edge behavior.
- Use a non-integer window size and inspect unused borders.
- Toggle linear scale mode on one texture only.
- Test 100%, 150%, and 200% desktop scaling.

What to Watch For

- Using nearest filtering for photographic assets.
- Assuming high-DPI means the same thing as logical presentation.
- Mixing output pixels into game-world measurements.
- Neglecting event coordinate conversion.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Sharpness is a policy choice. Declare logical resolution and filtering explicitly instead of relying on defaults that vary by asset and display.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the pixel-perfect scaling exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 12 — BUILD

Event Trace Lab

GOAL	STARTING POINT
Build a small diagnostic event tracer so you can see what SDL sends instead of guessing.	Run a normal event loop.

Build It

1. Log quit, key, mouse, and window events.
2. Print only meaningful fields.
3. Keep the tracer easy to enable/disable.
4. Resize and focus/unfocus the window.
5. Compare event-driven state with polled state.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
while (SDL_PollEvent(&event)) {
    switch (event.type) {
    case SDL_EVENT_QUIT:
        SDL_Log("quit");
        running = false;
        break;
    case SDL_EVENT_KEY_DOWN:
        SDL_Log("key scancode=%d", event.key.scancode);
        break;
    case SDL_EVENT_MOUSE_MOTION:
        SDL_Log("mouse %.1f %.1f", event.motion.x, event.motion.y);
        break;
    case SDL_EVENT_WINDOW_RESIZED:
        SDL_Log("resize %d %d", event.window.data1, event.window.data2);
        break;
    }
}
```

Expected Result Console output shows the sequence and payload of common events as they arrive.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Hold a key and compare discrete key events with continuous keyboard state.
- Move, resize, minimize, and refocus the window while logging event types.
- Filter the logger to one event family, then restore the full trace.
- Add timestamps so bursts of events can be recognized in the log.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 12 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Hold a key and inspect repeat behavior.
- Move the mouse rapidly and compare event volume.
- Resize continuously and observe how many events are generated.
- Use the tracer to diagnose one input bug in another lab.

What to Watch For

- Logging every event in a release build.
- Using event timestamps as a replacement for simulation time.
- Confusing one-shot events with continuous state.
- Ignoring window events that affect rendering assumptions.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle A trace tool is cheap instrumentation. Build it once and keep it available; visibility often saves more time than optimization.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the event trace exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 13 — BUILD

Continuous Input Lab

GOAL	STARTING POINT
Use keyboard and mouse state for continuous control while retaining events for one-shot actions.	Process events once per frame before reading current state.

Build It

1. Read SDL_GetKeyboardState.
2. Read SDL_GetMouseState.
3. Move with WASD using dt.
4. Use a mouse click event for a one-shot action.
5. Draw a cursor marker.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
const bool *keys = SDL_GetKeyboardState(NULL);
float speed = 220.0f;
if (keys[SDL_SCANCODE_W]) y -= speed * dt;
if (keys[SDL_SCANCODE_S]) y += speed * dt;
if (keys[SDL_SCANCODE_A]) x -= speed * dt;
if (keys[SDL_SCANCODE_D]) x += speed * dt;

float mx = 0.0f, my = 0.0f;
SDL_MouseButtonFlags buttons = SDL_GetMouseState(&mx, &my);
bool left_down = (buttons & SDL_BUTTON_LMASK) != 0;
```

Expected Result Movement is smooth while keys are held and the mouse position updates every frame.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Compare movement driven by key events with movement driven by SDL_GetKeyboardState.
- Normalize diagonal movement so two keys do not make the actor move faster.
- Use one mouse click as an edge-triggered action while cursor position remains state-based.
- Draw an on-screen input overlay and verify it agrees with the physical controls.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 13 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Compare state polling before and after event processing.
- Add diagonal movement normalization.
- Switch to relative mouse mode for a camera experiment.
- Use events for click edges and state for dragging.

What to Watch For

- Treating key-down events as continuous movement.

- Ignoring dt.
- Using keycodes where scancodes better represent physical controls.
- Mixing UI click logic directly into world movement code.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Use events for transitions and state queries for sustained conditions. The combination is clearer than forcing one model to do both jobs.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the continuous input exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 14 — BUILD

Gamepad Discovery Lab

GOAL	STARTING POINT
Discover connected gamepads safely and open the first one without assuming a device index.	Initialize SDL_INIT_GAMEPAD.

Build It

1. Ask SDL for the current gamepad IDs.
2. Open the first returned ID.
3. Log its name.
4. Handle the no-gamepad case.
5. Close the gamepad and free the ID list.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
int count = 0;
SDL_JoystickID *ids = SDL_GetGamepads(&count);
SDL_Gamepad *pad = NULL;
if (ids && count > 0) {
```

```
    pad = SDL_OpenGamepad(ids[0]);
    if (pad)
        SDL_Log("gamepad=%s", SDL_GetGamepadName(pad));
}
SDL_free(ids);

/* use pad while the application runs */

if (pad)
    SDL_CloseGamepad(pad);
```

Expected Result The program works whether zero, one, or several gamepads are connected.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Run with no controller connected and treat that as a normal state.
- Connect a controller after startup and inspect the device events you receive.
- Log the opened gamepad name and keep gameplay actions independent of the model name.
- Close and reopen the device to verify ownership and cleanup are explicit.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 14 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Unplug and reconnect while the program runs and inspect events.
- Map one button to the same action as a keyboard key.
- Query a stick axis and add a dead zone.
- Call SDL_GetTouchDevices on a touch-capable platform and compare discovery patterns.

What to Watch For

- Assuming device index is permanent identity.
- Forgetting to free returned ID arrays.
- Failing to handle disconnect events.
- Binding game logic directly to one physical device.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.

Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Build an input abstraction around actions such as Move, Jump, Confirm, and Cancel. Devices then become interchangeable sources.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the gamepad discovery exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 15 — BUILD

Fixed-Step Timing Lab

GOAL	STARTING POINT
Run simulation at a fixed update interval while rendering as often as the machine allows.	Start with a variable-dt loop that already works.

Build It

1. Accumulate real elapsed time.
2. Run zero or more fixed updates.
3. Keep the step constant.
4. Render after updates.
5. Clamp extreme frame delays.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
const double step = 1.0 / 120.0;
double accumulator = 0.0;
Uint64 previous = SDL_GetTicksNS();

Uint64 now = SDL_GetTicksNS();
double frame = (double)(now - previous) / 1e9;
previous = now;
if (frame > 0.25) frame = 0.25;
accumulator += frame;

while (accumulator >= step) {
    UpdateSimulation((float)step);
    accumulator -= step;
}
RenderFrame(renderer);
```

Expected Result Simulation behavior stays stable even if rendering cadence changes.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Run the same simulation with two different render rates while keeping the fixed step unchanged.
- Clamp a large elapsed-time spike and compare recovery with and without the clamp.
- Count how many fixed updates occur during a deliberately slow frame.
- Interpolate a visual value between simulation states and compare smoothness.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 15 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Change fixed rate from 60 to 120 Hz.
- Artificially stall one frame and inspect how many updates run afterward.
- Add interpolation between previous and current simulation states.
- Measure total update time per render frame.

What to Watch For

- The spiral of death when updates cannot catch up.
- Using unbounded catch-up after a long pause.
- Tying deterministic logic to render frame count.
- Using fixed step everywhere when variable dt is simpler and sufficient.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Fixed-step simulation is a tool, not a religion. Use it where determinism or stability matters and keep rendering decoupled.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the fixed-step timing exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 16 — BUILD

Audio Stream Lab

GOAL	STARTING POINT
Open the default playback device through SDL_AudioStream and queue prepared PCM data.	Have a small PCM buffer in the format described by SDL_AudioSpec.

Build It

1. Initialize SDL_INIT_AUDIO.
2. Describe your source format.
3. Open a default playback stream.
4. Resume the stream device.
5. Queue data and destroy the stream at shutdown.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
SDL_AudioSpec spec = {
    .format = SDL_AUDIO_F32,
    .channels = 2,
    .freq = 48000
};
SDL_AudioStream *stream = SDL_OpenAudioDeviceStream(
    SDL_AUDIO_DEVICE_DEFAULT_PLAYBACK,
    &spec, NULL, NULL);
if (!stream) {
    SDL_Log("audio: %s", SDL_GetError());
    return 1;
}
SDL_ResumeAudioStreamDevice(stream);
SDL_PutAudioStreamData(stream, samples, sample_bytes);

/* later */
SDL_DestroyAudioStream(stream);
```

Expected Result Queued PCM data plays through the default device while SDL owns conversion/device details behind the stream.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Start with a larger queued buffer, then reduce it carefully while listening for underruns.
- Pause and resume playback without recreating the audio stream.
- Queue short chunks repeatedly and monitor how much data remains queued.
- Change source volume in your sample data while keeping the device format unchanged.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 16 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Query queued byte count while sound plays.
- Reduce the amount queued ahead and listen for underruns.
- Adjust stream gain.
- Queue two buffers with different content and verify ordering.

What to Watch For

- Passing sample data in a format different from spec.
- Destroying or reusing a buffer too early when your own code still needs it.
- Doing file decoding in the real-time path.
- Choosing huge buffering just to hide design problems.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Audio latency is an end-to-end property. Measure from input action to audible result, not only the size of one buffer.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the audio stream exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 17 — BUILD

Mini Scene Integration Lab

GOAL	STARTING POINT
Combine timing, input, textures, sprites, and a HUD into one small scene without losing architectural clarity.	Reuse working pieces from earlier labs.

Build It

1. Create a GameState struct.
2. Handle input into intent/state.
3. Update position with dt.
4. Render background, actor, then UI.
5. Release every owned resource.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
typedef struct {
    float x, y;
    float speed;
    int score;
} GameState;

void Update(GameState *g, float dt, const bool *keys) {
    float dx = (keys[SDL_SCANCODE_D] ? 1.0f : 0.0f) -
               (keys[SDL_SCANCODE_A] ? 1.0f : 0.0f);
    g->x += dx * g->speed * dt;
}

void Render(SDL_Renderer *r, const GameState *g,
            SDL_Texture *player) {
    SDL_SetRenderDrawColor(r, 16, 24, 42, 255);
    SDL_RenderClear(r);
    SDL_FRect dst = {g->x, g->y, 48, 48};
    SDL_RenderTexture(r, player, NULL, &dst);
    DrawHUD(r, g->score);
    SDL_RenderPresent(r);
}
```

Expected Result A tiny but layered 2D application emerges from the same simple resource and frame-loop rules used throughout the book.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Add one second animated actor without changing the first actor component.
- Insert a camera offset and keep world coordinates separate from screen coordinates.
- Add a pause overlay that stops updates but continues processing window events.
- Count owned textures, fonts, and streams and verify every count returns to zero at shutdown.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 17 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Add a collectible and increment score.
- Add one animation clip.
- Cache HUD text until score changes.
- Record update and render time separately.

What to Watch For

- Turning GameState into a bag of renderer pointers.
- Letting rendering mutate simulation state.
- Loading assets inside DrawHUD or Render.
- Losing track of ownership as the demo grows.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Integration reveals architecture quality. When each subsystem has a narrow contract, adding one feature does not require touching everything else.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the mini scene integration exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 18 — BUILD

Failure Injection Lab

GOAL	STARTING POINT
Deliberately create failures and prove that diagnostics point to the actual stage that failed.	Use a copy of a working project so destructive experiments are safe.

Build It

1. Break one asset path.
2. Force one resource creation failure path.
3. Log SDL_GetError immediately.
4. Clean up already-created resources.
5. Restore the bug and confirm normal behavior returns.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
SDL_Texture *LoadTextureChecked(SDL_Renderer *r, const char *path) {
    SDL_Surface *s = SDL_LoadSurface(path);
    if (!s) {
        SDL_Log("LoadSurface(%s): %s", path, SDL_GetError());
        return NULL;
    }
    SDL_Texture *t = SDL_CreateTextureFromSurface(r, s);
    SDL_DestroySurface(s);
    if (!t)
        SDL_Log("CreateTextureFromSurface(%s): %s",
                path, SDL_GetError());
    return t;
}
```

Expected Result A broken asset path produces a precise message containing the failed operation and path instead of a vague “SDL failed.”

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Break one image path, one font path, and one CMake dependency separately.
- Force a resource creation failure after another resource has already succeeded.
- Record the first failing operation before any later cleanup call can overwrite the error context.
- Restore the failure and confirm the same executable returns to the success path without code changes elsewhere.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 18 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Pass a nonexistent file.
- Temporarily remove SDL3 from the runtime search path.
- Simulate a NULL texture before rendering.
- Add assertions for invariants that must never be false in development builds.

What to Watch For

- Logging too late after another SDL call changed the error string.
- Continuing with NULL resources.
- Returning early without cleanup.
- Treating every warning as fatal or every failure as recoverable.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Good error messages are part of the interface of your program—to yourself first, and eventually to users and support teams.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the failure injection exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 19 — BUILD

Release Linkage Lab

GOAL	STARTING POINT
Produce both shared-oriented and static-oriented builds through CMake targets without manual include paths.	Have an SDL3 installation that provides the desired target variants.

Build It

1. Keep source target definitions identical.
2. Select the SDL imported target appropriate to the package.
3. Build Release with optimization.
4. Inspect runtime dependencies.
5. Package required licenses and libraries.

Copy-Ready Focus Listing

CMAKE • COPY-READY

```
cmake_minimum_required(VERSION 3.25)
project(SDL3Release LANGUAGES C)
set(CMAKE_C_STANDARD 23)
find_package(SDL3 CONFIG REQUIRED)
add_executable(app src/main.c)

if(TARGET SDL3::SDL3-shared)
    target_link_libraries(app PRIVATE SDL3::SDL3-shared)
else()
    target_link_libraries(app PRIVATE SDL3::SDL3)
endif()

target_compile_options(app PRIVATE
    $<$<C_COMPILER_ID:GNU>:-O2 -Wall -Wextra>)
```

Expected Result The build remains target-based: include paths and link details arrive from SDL's CMake package instead of being hard-coded.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Build shared and static variants when the installed SDL package provides both targets.
- Inspect runtime dependencies from a clean release directory.
- Move the release to a clean test location and run it without development tools.
- Compare binary size, required companion libraries, and startup behavior.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 19 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Inspect ldd/otool/dependency output for the shared build.
- Compare executable size with a static-capable package.
- Run the release on a clean machine or VM.
- Remove one required shared library to validate your packaging checks.

What to Watch For

- Assuming “static” means one file on every platform.
- Forgetting transitive system/runtime dependencies.
- Shipping debug assets or symbols unintentionally.
- Using compiler-specific flags without generator expressions.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.

Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Deployment is part of correctness. A program that works only on the development machine is not yet a finished build.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the release linkage exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

GUIDED LAB 20 — BUILD

Cross-Platform Paths Lab

GOAL	STARTING POINT
Stop depending on the current working directory and use SDL filesystem helpers for assets and writable user data.	Choose one read-only asset and one writable settings file.

Build It

1. Query SDL_GetBasePath for application resources.
2. Query SDL_GetPrefPath for writable user data.
3. Build paths explicitly.
4. Free the preference path when finished.
5. Test from a different working directory.

Copy-Ready Focus Listing

C23 • COPY-READY

```c
const char *base = SDL_GetBasePath();
if (!base) {
    SDL_Log("base path: %s", SDL_GetError());
    return 1;
}

char *pref = SDL_GetPrefPath("AymanAlheraki", "SDL3Lab");
if (!pref) {
    SDL_Log("pref path: %s", SDL_GetError());
    return 1;
}

SDL_Log("assets: %s", base);
SDL_Log("writable: %s", pref);

/* write settings under pref, then */
SDL_free(pref);
```

Expected Result The same executable can find its application resources and a writable per-user location without assuming where it was launched from.

Before You Move On

- Predict one visible or logged result before you change a value, flag, path, timing parameter, or input condition.
- Restore the original value and confirm the behavior returns; this proves you understood cause and effect rather than coincidence.
- Save the working state as a small commit or separate target so the experiment remains reproducible later.

Ownership Contract Before leaving the lab, point to every SDL object that was created and name the exact cleanup function or owning subsystem responsible for releasing it.

Try These Variations

- Launch from CLion, a terminal, and a file manager and confirm resource discovery is stable.
- Store one writable preference value and verify it appears in the platform user-data location.
- Change the process working directory before launch and prove the application still finds packaged assets.
- Log base path, preference path, and platform name once at startup for diagnostic evidence.

IDE Verification

Run the experiment from both CLion and a clean terminal build. Keep warnings enabled, make one change at a time, and preserve one visible result that proves the behavior you intended.

GUIDED LAB 20 — OBSERVE

Observe, Measure, and Extend

Do not stop when the first version works. The useful knowledge appears when you vary one assumption and predict the consequence before you run the program. Record what changed, what did not change, and which layer of the system was responsible.

Run These Experiments

- Launch from the terminal and from the file manager.
- Move the release folder and rerun.
- Store a small settings file under pref.
- Print SDL_GetPlatform and compare results across systems.

What to Watch For

- Using the current working directory as an asset contract.
- Writing user data beside the executable on protected locations.
- Forgetting SDL_GetPrefPath returns allocated memory.
- Hard-coding Windows or POSIX separators into every path.

Debug Checkpoint

CHECK	WHAT SUCCESS LOOKS LIKE
Return values	Every creation or state-changing call that can fail is checked at the point of use.
Ownership	You can name the function responsible for destroying every SDL resource created in the lab.
State	You can identify whether the observed value belongs to input, simulation, renderer, window, or resource state.
Timing	Any motion, animation, or audio behavior has an explicit time/buffering model instead of relying on frame count.
Re-run	The lab still works after a clean rebuild and from a fresh launch environment.

Performance / Design Note

Keep This Principle Portability improves when platform differences are isolated behind small path, input, and packaging boundaries rather than scattered across gameplay code.

Checkpoint Questions

- Can you explain the lifetime of every resource used here without looking at the code?
- Which value would you log first if the visible result were wrong?
- Which part belongs outside the hot per-frame path?
- What single change would make this lab reusable in a larger application?

Evidence to Capture

- One screenshot or visual frame that proves the expected output, with the important state clearly visible.
- One short console/log excerpt that records the relevant event, value, resource transition, or error path.
- A two-sentence note: what you changed, what changed as a result, and which subsystem owns that behavior.

Transfer This Skill

Treat the cross-platform paths exercise as a reusable debugging pattern: isolate one contract, make its state observable, vary one assumption, and keep the evidence that explains why the result changed.

Capstone Challenge — Combine the Whole Book

Create one polished 2D application that uses a resizable high-pixel-density window, logical presentation, texture assets, sprite animation, keyboard plus gamepad input, a cached text HUD, one audio stream, and a clean release build. Keep the implementation intentionally small: the challenge is not feature count, but whether every subsystem has a clear owner, update path, rendering responsibility, and failure path.

- Measure update time, render time, and the longest frame observed during a normal session.
- Run the executable from a clean build directory and from a packaged release folder.
- Trigger at least three controlled failures: missing asset, unavailable gamepad, and invalid writable path.
- Write a one-page architecture note explaining who owns the window, renderer, textures, font/text cache, audio stream, and game state.
- Only after correctness is stable, profile one real bottleneck and improve it without changing program behavior.

Definition of Done A reader should be able to clone or copy the project, configure it with CMake, build it with GCC 16.1 in CLion or the terminal, run it without hidden machine-specific paths, and understand every resource lifetime from creation to destruction.

APPENDIX G — COPY-READY TEMPLATES

Eight Templates to Start Real Projects Faster

These pages are intentionally source-heavy. Printed line numbers are omitted from copy-ready listings, so readers can select the code itself and paste clean source text into CLion or another IDE. Each template is small enough to understand, yet structured enough to reuse in a real CMake project.

TEMPLATE 1

Minimal SDL3 C23 Application

Use this as the smallest complete starting point for a new renderer-based experiment. It owns exactly one window and one renderer and has one obvious cleanup path.

Source

C23 • COPY-READY

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

int main(void) {
    if (!SDL_Init(SDL_INIT_VIDEO)) {
        SDL_Log("SDL_Init: %s", SDL_GetError());
        return 1;
    }

    SDL_Window *window = NULL;
    SDL_Renderer *renderer = NULL;
    if (!SDL_CreateWindowAndRenderer(
            "SDL3 C23 Starter", 960, 540,
            SDL_WINDOW_RESIZABLE | SDL_WINDOW_HIGH_PIXEL_DENSITY,
            &window, &renderer)) {
        SDL_Log("CreateWindowAndRenderer: %s", SDL_GetError());
        SDL_Quit();
        return 1;
    }

    bool running = true;
    SDL_Event event;
    while (running) {
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_EVENT_QUIT ||
                event.type == SDL_EVENT_WINDOW_CLOSE_REQUESTED)
                running = false;
```

```
        }

        SDL_SetRenderDrawColor(renderer, 14, 22, 38, 255);
        SDL_RenderClear(renderer);
        SDL_RenderPresent(renderer);
    }

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Use It Correctly

- Paste as src/main.c in a project already linked to SDL3::SDL3.
- Add one new concept at a time; do not turn the starter into a framework before you need one.
- If creation fails, the error is logged at the exact failing stage.

Copy/Paste Note Printed line numbers are intentionally omitted. Copy the code listing directly into your IDE. After pasting, run your formatter and compile with warnings enabled before adding more code.

When to Use This Template

- Starting a new rendering experiment with one window and one renderer.
- Testing a new SDL API in the smallest useful lifetime context.
- Creating a reproducible bug case before adding framework code.

Customize Before Shipping

- Choose the real window title and initial dimensions.
- Add only the SDL subsystems the program actually uses.
- Keep ownership explicit as additional textures, fonts, or streams are introduced.

Verification Checklist

- **Paste the code listing into your source file, run your formatter, and compile with warnings enabled.**
- Run once from outside CLion so hidden IDE working-directory assumptions are exposed.
- Deliberately fail one dependency or resource path and confirm the diagnostic points to the exact failing operation.

Copy-ready means more than selectable text: the snippet should have an explicit lifetime, clear failure behavior, no printed line numbers mixed into the source, and no hidden dependency on an IDE-specific path.

TEMPLATE 2

Structured Frame Loop

Use this skeleton when the program has enough behavior that event handling, state update, and rendering should stop living in one function.

Source

C23 • COPY-READY

```c
typedef struct App {
    bool running;
    float x;
    float velocity;
} App;

static void HandleEvents(App *app) {
    SDL_Event e;
    while (SDL_PollEvent(&e)) {
        if (e.type == SDL_EVENT_QUIT)
            app->running = false;
    }
}

static void Update(App *app, float dt) {
    app->x += app->velocity * dt;
}

static void Render(SDL_Renderer *r, const App *app) {
    SDL_SetRenderDrawColor(r, 12, 18, 32, 255);
    SDL_RenderClear(r);
    SDL_FRect box = {app->x, 220.0f, 48.0f, 48.0f};
    SDL_SetRenderDrawColor(r, 80, 160, 255, 255);
```

```
    SDL_RenderFillRect(r, &box);
    SDL_RenderPresent(r);
}

/* inside main loop */
HandleEvents(&app);
Update(&app, dt);
Render(renderer, &app);
```

Use It Correctly

- Keep Update free of renderer calls.
- Keep Render observational: it should not change simulation rules.
- Add timing instrumentation around Update and Render separately.

Copy/Paste Note Printed line numbers are intentionally omitted. Copy the code listing directly into your IDE. After pasting, run your formatter and compile with warnings enabled before adding more code.

When to Use This Template

- Separating event handling, state update, and rendering in a small project.
- Making timing and renderer boundaries obvious before the codebase grows.
- Adding profiling around Update and Render without mixing responsibilities.

Customize Before Shipping

- Keep input collection outside gameplay mutation where possible.
- Pass elapsed time explicitly into update code.
- Present once after all draw calls for the frame are complete.

Verification Checklist

- **Paste the code listing into your source file, run your formatter, and compile with warnings enabled.**
- Run once from outside CLion so hidden IDE working-directory assumptions are exposed.
- Deliberately fail one dependency or resource path and confirm the diagnostic points to the exact failing operation.

Copy-ready means more than selectable text: the snippet should have an explicit lifetime, clear failure behavior, no printed line numbers mixed into the source, and no hidden dependency on an IDE-specific path.

TEMPLATE 3

Texture Loader Helper

Use one loader contract everywhere so path errors, surface lifetime, and texture creation are handled consistently.

Source

C23 • COPY-READY

```c
static SDL_Texture *LoadTexture(
    SDL_Renderer *renderer,
    const char *path)
{
    SDL_Surface *surface = SDL_LoadSurface(path);
    if (!surface) {
        SDL_Log("SDL_LoadSurface(%s): %s",
                path, SDL_GetError());
        return NULL;
    }

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

    if (!texture) {
        SDL_Log("SDL_CreateTextureFromSurface(%s): %s",
                path, SDL_GetError());
        return NULL;
    }

    return texture;
}

/* startup */
SDL_Texture *player = LoadTexture(renderer,
                                  "assets/player.png");
if (!player) {
    /* clean up and stop startup */
```

```
}

/* shutdown */
SDL_DestroyTexture(player);
```

Use It Correctly

- The surface is temporary CPU-side data; the returned texture becomes the long-lived renderer resource.
- Never continue startup with a NULL required asset.
- For many assets, place returned textures in an explicit asset/resource owner.

Copy/Paste Note Printed line numbers are intentionally omitted. Copy the code listing directly into your IDE. After pasting, run your formatter and compile with warnings enabled before adding more code.

When to Use This Template

- Centralizing image-load errors and surface-to-texture conversion.
- Giving asset code one clear ownership contract.
- Avoiding repeated path, error, and cleanup code across callers.

Customize Before Shipping

- Decide whether your project needs SDL_LoadSurface or a dedicated image add-on path.
- Define the asset-root policy instead of depending on the working directory.
- Decide whether missing assets are fatal or recoverable for your application.

Verification Checklist

- **Paste the code listing into your source file, run your formatter, and compile with warnings enabled.**
- Run once from outside CLion so hidden IDE working-directory assumptions are exposed.
- Deliberately fail one dependency or resource path and confirm the diagnostic points to the exact failing operation.

Copy-ready means more than selectable text: the snippet should have an explicit lifetime, clear failure behavior, no printed line numbers mixed into the source, and no hidden dependency on an IDE-specific path.

TEMPLATE 4

Sprite Animation Component

This small component keeps animation timing separate from movement/game rules and produces a source rectangle for SDL_RenderTexture.

Source

C23 • COPY-READY

```c
typedef struct Animation {
    int first_frame;
    int frame_count;
    int current;
    int frame_w;
    int frame_h;
    int columns;
    float seconds_per_frame;
    float accumulator;
} Animation;

static void AnimationUpdate(Animation *a, float dt) {
    a->accumulator += dt;
    while (a->accumulator >= a->seconds_per_frame) {
        a->accumulator -= a->seconds_per_frame;
        a->current = (a->current + 1) % a->frame_count;
    }
}

static SDL_FRect AnimationSource(const Animation *a) {
    int frame = a->first_frame + a->current;
    int col = frame % a->columns;
    int row = frame / a->columns;
    return (SDL_FRect){
        (float)(col * a->frame_w),
        (float)(row * a->frame_h),
        (float)a->frame_w,
        (float)a->frame_h
    };
}
```

Use It Correctly

- Game logic selects the active animation; the animation component only advances it.
- Subtract frame duration instead of resetting the accumulator so time is not silently lost.
- Use a destination rectangle independent of source frame size when scaling sprites.

Copy/Paste Note Printed line numbers are intentionally omitted. Copy the code listing directly into your IDE. After pasting, run your formatter and compile with warnings enabled before adding more code.

When to Use This Template

- Keeping frame timing separate from movement and gameplay rules.
- Reusing one animation component across multiple actors.
- Making source-rectangle selection deterministic and testable.

Customize Before Shipping

- Define clip ranges from your actual sprite atlas.
- Choose looping behavior per clip instead of globally.
- Keep movement state responsible for choosing a clip, not for advancing its frames.

Verification Checklist

- **Paste the code listing into your source file, run your formatter, and compile with warnings enabled.**
- Run once from outside CLion so hidden IDE working-directory assumptions are exposed.
- Deliberately fail one dependency or resource path and confirm the diagnostic points to the exact failing operation.

Copy-ready means more than selectable text: the snippet should have an explicit lifetime, clear failure behavior, no printed line numbers mixed into the source, and no hidden dependency on an IDE-specific path.

TEMPLATE 5

Cached HUD Text Helper

Use this pattern when text changes occasionally, such as score, health, mode names, or diagnostics. Rebuild only when the string changes.

Source

C23 • COPY-READY

```c
#include <SDL3_ttf/SDL_ttf.h>

static SDL_Texture *MakeTextTexture(
    SDL_Renderer *renderer,
    TTF_Font *font,
    const char *text,
    SDL_Color color)
{
    SDL_Surface *surface =
        TTF_RenderText_Blended(font, text, 0, color);
    if (!surface) {
        SDL_Log("TTF_RenderText_Blended: %s", SDL_GetError());
        return NULL;
    }

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

    if (!texture)
        SDL_Log("CreateTextureFromSurface: %s", SDL_GetError());
    return texture;
}

/* when score changes */
SDL_DestroyTexture(score_text);
char buffer[64];
SDL_snprintf(buffer, sizeof(buffer), "SCORE %d", score);
score_text = MakeTextTexture(renderer, font, buffer,
    (SDL_Color){255, 255, 255, 255});
```

Use It Correctly

- Open fonts at startup and close them at shutdown.
- Cache text textures whose contents are unchanged.
- Keep font licensing and redistribution requirements with the shipped assets.

Copy/Paste Note Printed line numbers are intentionally omitted. Copy the code listing directly into your IDE. After pasting, run your formatter and compile with warnings enabled before adding more code.

When to Use This Template

- Rendering score, status, debug labels, or menu text that changes occasionally.
- Avoiding surface and texture creation every frame.
- Centralizing text-texture ownership and invalidation.

Customize Before Shipping

- Choose fonts and licenses appropriate for redistribution.
- Rebuild cached text only when the source string or style changes.
- Store size information with the cache if layout needs exact text dimensions.

Verification Checklist

- **Paste the code listing into your source file, run your formatter, and compile with warnings enabled.**
- Run once from outside CLion so hidden IDE working-directory assumptions are exposed.
- Deliberately fail one dependency or resource path and confirm the diagnostic points to the exact failing operation.

Copy-ready means more than selectable text: the snippet should have an explicit lifetime, clear failure behavior, no printed line numbers mixed into the source, and no hidden dependency on an IDE-specific path.

TEMPLATE 6

SDL3 Audio Stream Starter

This helper opens the default playback device through one SDL_AudioStream and makes queued PCM ownership explicit.

Source

C23 • COPY-READY

```c
static SDL_AudioStream *OpenPlaybackStream(void) {
    SDL_AudioSpec spec = {
        .format = SDL_AUDIO_F32,
        .channels = 2,
        .freq = 48000
    };

    SDL_AudioStream *stream = SDL_OpenAudioDeviceStream(
        SDL_AUDIO_DEVICE_DEFAULT_PLAYBACK,
        &spec,
        NULL,
        NULL);
    if (!stream) {
        SDL_Log("OpenAudioDeviceStream: %s", SDL_GetError());
        return NULL;
    }

    if (!SDL_ResumeAudioStreamDevice(stream)) {
        SDL_Log("ResumeAudioStreamDevice: %s", SDL_GetError());
        SDL_DestroyAudioStream(stream);
        return NULL;
    }
    return stream;
}

/* queue already-prepared float stereo PCM */
if (!SDL_PutAudioStreamData(stream, samples, byte_count))
    SDL_Log("PutAudioStreamData: %s", SDL_GetError());

/* shutdown */
SDL_DestroyAudioStream(stream);
```

Use It Correctly

- The spec describes the application side of the stream.
- Decode files outside latency-sensitive playback paths.
- Tune queued-ahead audio by measurement on target hardware, not by one universal constant.

Copy/Paste Note Printed line numbers are intentionally omitted. Copy the code listing directly into your IDE. After pasting, run your formatter and compile with warnings enabled before adding more code.

When to Use This Template

- Opening a simple playback path through SDL_AudioStream.
- Feeding decoded or generated PCM data to the default output device.
- Creating a small test bed for latency and queue-size experiments.

Customize Before Shipping

- Match the SDL_AudioSpec to the data your application actually supplies.
- Keep decoding/file I/O outside the real-time-sensitive playback path.
- Handle device loss and stream failure as normal runtime states.

Verification Checklist

- **Paste the code listing into your source file, run your formatter, and compile with warnings enabled.**
- Run once from outside CLion so hidden IDE working-directory assumptions are exposed.
- Deliberately fail one dependency or resource path and confirm the diagnostic points to the exact failing operation.

Copy-ready means more than selectable text: the snippet should have an explicit lifetime, clear failure behavior, no printed line numbers mixed into the source, and no hidden dependency on an IDE-specific path.

TEMPLATE 7

Portable CMake Target

This target keeps C23 requirements, warnings, SDL3 discovery, and shared/static selection in the build description instead of editor settings.

Source

CMAKE • COPY-READY

```cmake
cmake_minimum_required(VERSION 3.25)
project(SDL3App VERSION 1.0 LANGUAGES C)

set(CMAKE_C_STANDARD 23)
set(CMAKE_C_STANDARD_REQUIRED ON)
set(CMAKE_C_EXTENSIONS OFF)

find_package(SDL3 CONFIG REQUIRED)

add_executable(SDL3App
    src/main.c
    src/game.c
    src/assets.c
)

target_include_directories(SDL3App PRIVATE include)

target_compile_options(SDL3App PRIVATE
    $<$<C_COMPILER_ID:GNU>:
        -Wall -Wextra -Wpedantic -Wconversion -Wshadow>)

if(TARGET SDL3::SDL3-shared)
    target_link_libraries(SDL3App PRIVATE SDL3::SDL3-shared)
else()
    target_link_libraries(SDL3App PRIVATE SDL3::SDL3)
endif()

install(TARGETS SDL3App RUNTIME DESTINATION bin)
install(DIRECTORY assets/ DESTINATION assets)
```

Use It Correctly

- CLion should import this target instead of duplicating compiler/link settings in the IDE.
- Use CMakePresets.json for Debug/Release configuration rather than hand-editing build directories.
- Test installation/package layout separately from the developer build tree.

Copy/Paste Note Printed line numbers are intentionally omitted. Copy the code listing directly into your IDE. After pasting, run your formatter and compile with warnings enabled before adding more code.

When to Use This Template

- Keeping C23, warnings, SDL discovery, and target options in one build description.
- Sharing the same build logic between CLion and command-line workflows.
- Selecting imported SDL targets instead of hard-coded include or library paths.

Customize Before Shipping

- Add project source files and include directories through target_* commands.
- Keep developer-only sanitizer flags in a dedicated Debug/sanitized profile.
- Test installation or packaging rules from a clean Release build directory.

Verification Checklist

- **Paste the code listing into your source file, run your formatter, and compile with warnings enabled.**
- Run once from outside CLion so hidden IDE working-directory assumptions are exposed.
- Deliberately fail one dependency or resource path and confirm the diagnostic points to the exact failing operation.

Copy-ready means more than selectable text: the snippet should have an explicit lifetime, clear failure behavior, no printed line numbers mixed into the source, and no hidden dependency on an IDE-specific path.

TEMPLATE 8

Cross-Platform Resource Paths

Use SDL filesystem helpers to separate application resources from writable per-user data and avoid depending on the launch directory.

Source

C23 • COPY-READY

```c
static bool PrintApplicationPaths(void) {
    const char *base = SDL_GetBasePath();
    if (!base) {
        SDL_Log("SDL_GetBasePath: %s", SDL_GetError());
        return false;
    }

    char *pref = SDL_GetPrefPath(
        "AymanAlheraki",
        "SDL3Example");
    if (!pref) {
        SDL_Log("SDL_GetPrefPath: %s", SDL_GetError());
        return false;
    }

    SDL_Log("platform: %s", SDL_GetPlatform());
    SDL_Log("base path: %s", base);
    SDL_Log("preference path: %s", pref);

    /* copy pref into app-owned storage if needed */
    SDL_free(pref);
    return true;
}
```

Use It Correctly

- Use the base path for packaged read-only resources when appropriate.
- Use the preference path for saves/settings rather than the executable directory.
- Launch from different working directories during testing to expose hidden path assumptions.

Copy/Paste Note Printed line numbers are intentionally omitted. Copy the code listing directly into your IDE. After pasting, run your formatter and compile with warnings enabled before adding more code.

When to Use This Template

- Finding packaged read-only assets independently of the process working directory.
- Finding a writable per-user location for settings, saves, and caches.
- Making platform path differences explicit behind a very small helper.

Customize Before Shipping

- Define which files are application assets and which are mutable user data.
- Join path components safely instead of concatenating assumptions throughout the program.
- Test launches from directories that differ from the executable location.

Verification Checklist

- **Paste the code listing into your source file, run your formatter, and compile with warnings enabled.**
- Run once from outside CLion so hidden IDE working-directory assumptions are exposed.
- Deliberately fail one dependency or resource path and confirm the diagnostic points to the exact failing operation.

Copy-ready means more than selectable text: the snippet should have an explicit lifetime, clear failure behavior, no printed line numbers mixed into the source, and no hidden dependency on an IDE-specific path.

Final Perspective — Make the System Observable

SDL3 is easiest to master when every layer has a visible contract: resources have owners, events become state, state becomes an update, rendering consumes that state, and presentation happens once the frame is complete. Keep those boundaries explicit and debugging becomes dramatically simpler.

Your Next Project

- Build one small 2D application that uses a window, renderer, texture, animation, input, text, and audio stream.
- Keep platform paths, build configuration, and release packaging independent of CLion-specific assumptions.
- Profile only after correctness is visible; optimize measured bottlenecks rather than guessed ones.
- Return to the copy-ready templates as starting points, not as architecture that must be preserved forever.

The goal of this book is not to memorize SDL3. It is to make the library predictable enough that you can experiment confidently, see what each layer does, and build your own graphics systems in C23 without unnecessary ceremony.

APPENDIX H — GRAPHICS COOKBOOK

Six Focused Recipes You Can Paste, Run, and Extend

These recipes are intentionally small. Each one isolates a graphics-programming problem, uses current SDL3-style APIs, omits printed line numbers from the source listing, and gives you concrete experiments to perform after the first successful run.

Copy/Paste Rule Copy the source listing into your C23 project, run your formatter, compile with warnings enabled, and connect the focused helper to objects that are already valid in your application.

COOKBOOK 01

Coordinate-Aware Mouse Picking

When logical presentation, scaling, or viewports are active, raw window coordinates and render coordinates are not the same thing. Convert the event before using it for picking or drawing.

Copy-Ready Focus Code

C23 • COPY-READY

```c
static void HandleEvent(SDL_Renderer *renderer,
                        SDL_Event *e,
                        float *pick_x, float *pick_y)
{
    if (e->type == SDL_EVENT_MOUSE_MOTION ||
        e->type == SDL_EVENT_MOUSE_BUTTON_DOWN) {
        if (!SDL_ConvertEventToRenderCoordinates(renderer, e)) {
            SDL_Log("coordinate conversion: %s", SDL_GetError());
            return;
        }
    }

    if (e->type == SDL_EVENT_MOUSE_BUTTON_DOWN) {
        *pick_x = e->button.x;
        *pick_y = e->button.y;
    }
}

static void DrawCrosshair(SDL_Renderer *renderer,
                          float x, float y)
{
    SDL_SetRenderDrawColor(renderer, 255, 210, 30, 255);
    SDL_RenderLine(renderer, x - 8.0f, y, x + 8.0f, y);
    SDL_RenderLine(renderer, x, y - 8.0f, x, y + 8.0f);
}
```

Expected Result

Picking remains stable even when the window is resized or letterboxed.

Experiments That Teach the API

- Enable SDL_SetRenderLogicalPresentation before testing.
- Resize to several aspect ratios and click the same logical object.
- Log both the converted position and the object that was selected.

Debug Habit Change one assumption at a time. When behavior is wrong, log the exact failing SDL call immediately, then verify coordinates, lifetime, target/state, and asset/path assumptions before adding more code.

Make It Yours

- Rename the helper around the vocabulary of your own project instead of preserving tutorial names.

COOKBOOK 02

Camera Scrolling Without Moving the World

Keep world coordinates stable. The camera is an offset applied when the world is converted into destination rectangles for rendering.

Copy-Ready Focus Code

C23 • COPY-READY

```c
typedef struct Camera {
    float x, y;
    float w, h;
} Camera;

static void Follow(Camera *c, float px, float py,
                   float world_w, float world_h)
{
    c->x = px - c->w * 0.5f;
    c->y = py - c->h * 0.5f;
    if (c->x < 0.0f) c->x = 0.0f;
    if (c->y < 0.0f) c->y = 0.0f;
    if (c->x > world_w - c->w) c->x = world_w - c->w;
    if (c->y > world_h - c->h) c->y = world_h - c->h;
}

static void DrawWorldTexture(SDL_Renderer *r, SDL_Texture *tex,
                             SDL_FRect world_rect, Camera c)
{
    SDL_FRect dst = {
        world_rect.x - c.x,
        world_rect.y - c.y,
        world_rect.w, world_rect.h
    };
    SDL_RenderTexture(r, tex, NULL, &dst);
}
```

Expected Result

The actor moves through world coordinates while the camera changes only the rendered offset.

Experiments That Teach the API

- Draw the camera bounds as a debug rectangle.
- Add a dead zone before the camera begins following the actor.
- Clamp to a world smaller than the viewport and decide how you want that case to behave.

Debug Habit Change one assumption at a time. When behavior is wrong, log the exact failing SDL call immediately, then verify coordinates, lifetime, target/state, and asset/path assumptions before adding more code.

Make It Yours

- Rename the helper around the vocabulary of your own project instead of preserving tutorial names.
- Keep the function narrow enough that its inputs, outputs, state changes, and owned resources can be explained in one minute.
- After it works, measure one real cost or failure mode before deciding whether another abstraction is useful.

Performance Lens

- Camera subtraction belongs at the render boundary; keep simulation coordinates in world space so collision, AI, and saved state do not depend on the viewport.
- For large worlds, combine the camera rectangle with culling so objects that cannot intersect the visible region are skipped before expensive rendering work.
- Measure camera math only after profiling. A few additions and subtractions are usually cheaper than extra abstractions, allocations, or per-frame container churn.

Failure Modes to Recognize

- Applying the camera offset to the actor state each frame causes world coordinates to drift and eventually breaks collision or save/load logic.
- Forgetting to clamp at map edges exposes empty space; clamping too early can also prevent a deliberate overscan or screen-shake effect.
- Mixing logical-presentation coordinates with raw window pixels makes picking and camera-centered UI disagree after resize or DPI changes.

CLion Verification

- Run with the camera stationary first, then move only the actor, then enable following. This isolates whether a defect belongs to simulation or rendering.
- Watch world position and screen position in separate debugger variables; their difference should equal the camera transform you intended.

COOKBOOK 03

Time-Based Sprite Atlas Animation

Animation speed should be controlled by elapsed time, not by how many frames the GPU happens to render.

Copy-Ready Focus Code

C23 • COPY-READY

```c
typedef struct Animation {
    int first;
    int count;
    int current;
    int columns;
    float seconds_per_frame;
    float accumulator;
    float frame_w, frame_h;
} Animation;

static void AnimUpdate(Animation *a, float dt)
{
    a->accumulator += dt;
    while (a->accumulator >= a->seconds_per_frame) {
        a->accumulator -= a->seconds_per_frame;
        a->current = (a->current + 1) % a->count;
    }
}

static SDL_FRect AnimSource(const Animation *a)
{
    const int frame = a->first + a->current;
    const int col = frame % a->columns;
    const int row = frame / a->columns;
    return (SDL_FRect){ col * a->frame_w, row * a->frame_h,
                        a->frame_w, a->frame_h };
}
```

Expected Result

The same clip advances at the same real-time speed even when rendering performance changes.

Experiments That Teach the API

- Render at deliberately different frame rates and compare animation speed.
- Make one clip loop and another stop on its final frame.
- Keep the animation component unaware of player velocity or input.

Debug Habit Change one assumption at a time. When behavior is wrong, log the exact failing SDL call immediately, then verify coordinates, lifetime, target/state, and asset/path assumptions before adding more code.

Make It Yours

- Rename the helper around the vocabulary of your own project instead of preserving tutorial names.
- Keep the function narrow enough that its inputs, outputs, state changes, and owned resources can be explained in one minute.
- After it works, measure one real cost or failure mode before deciding whether another abstraction is useful.

Performance Lens

- Animation update should be O(1) per animated object: accumulate elapsed time, advance only when a frame boundary is crossed, and compute one source rectangle.
- Do not rebuild textures or slice sprite images every frame. Keep the atlas resident and change only the source rectangle passed to the renderer.
- If hundreds of actors share one clip, measure update cost before introducing a scheduler; simple contiguous animation state is often cache-friendly enough.

Failure Modes to Recognize

- Advancing one frame per render makes animation speed depend on FPS and causes visibly different motion on fast and slow machines.
- Using a single if instead of a while when a very large dt arrives can leave animation permanently behind after a debugger pause.
- Letting movement logic directly change frame indices couples two systems and makes blending, one-shot clips, and replay behavior harder to reason about.

CLion Verification

- Force the renderer to several frame rates and confirm the clip duration in real seconds does not change.
- Pause in the debugger for one second, resume, and inspect whether the accumulator policy behaves exactly as your application requires.

COOKBOOK 04

Render-to-Texture Mini-Map

A render target lets you compose a small view once, then use the result like any other texture. Always restore the window target with NULL before drawing the final frame.

Copy-Ready Focus Code

C23 • COPY-READY

```c
static SDL_Texture *CreateMiniMap(SDL_Renderer *r)
{
    return SDL_CreateTexture(r, SDL_PIXELFORMAT_RGBA8888,
                             SDL_TEXTUREACCESS_TARGET, 256, 144);
}

static bool DrawMiniMap(SDL_Renderer *r, SDL_Texture *map)
{
    if (!SDL_SetRenderTarget(r, map)) return false;

    SDL_SetRenderDrawColor(r, 18, 28, 44, 255);
    SDL_RenderClear(r);
    /* draw simplified world markers here */

    if (!SDL_SetRenderTarget(r, NULL)) return false;

    const SDL_FRect dst = { 680.0f, 20.0f, 256.0f, 144.0f };
    SDL_RenderTexture(r, map, NULL, &dst);
    return true;
}

/* shutdown */
/* SDL_DestroyTexture(mini_map); */
```

Expected Result

The small map is composed off-screen and appears as a normal texture in the final window frame.

Experiments That Teach the API

- Deliberately omit the NULL restore once so you recognize the failure mode.
- Render only simple markers into the mini-map instead of the full scene.
- Update the mini-map less frequently than the main view and measure the difference.

Debug Habit Change one assumption at a time. When behavior is wrong, log the exact failing SDL call immediately, then verify coordinates, lifetime, target/state, and asset/path assumptions before adding more code.

Make It Yours

- Rename the helper around the vocabulary of your own project instead of preserving tutorial names.
- Keep the function narrow enough that its inputs, outputs, state changes, and owned resources can be explained in one minute.
- After it works, measure one real cost or failure mode before deciding whether another abstraction is useful.

Performance Lens

- A render target is valuable when its result can be reused, scaled, filtered, or updated less often than the main frame; otherwise it may only add bandwidth and state changes.
- Keep mini-map resolution intentionally small. Rendering a 256x144 target can be far cheaper than duplicating a full-resolution scene pass.
- Measure target switches and overdraw on the hardware you care about; the best update frequency may be every frame, every few frames, or only when world state changes.

Failure Modes to Recognize

- Forgetting SDL_SetRenderTarget(renderer, NULL) leaves later drawing on the off-screen texture, making the main window appear frozen or empty.
- Creating or destroying the target every frame turns a reusable GPU resource into avoidable allocation churn.
- Sampling a target while it is still the active destination creates a feedback dependency that should be redesigned rather than guessed around.

CLion Verification

- Break immediately before and after the target restore and inspect which texture is active; the state transition should be obvious.
- Temporarily clear the mini-map target to a loud diagnostic color so target ownership and update frequency are visible on screen.

COOKBOOK 05

A Tiny Frame-Time Overlay

SDL3 includes a debug text renderer that is useful for instrumentation. Pair it with SDL_GetTicksNS so performance evidence is visible while you experiment.

Copy-Ready Focus Code

C23 • COPY-READY

```c
typedef struct FrameMeter {
    Uint64 previous_ns;
    float frame_ms;
} FrameMeter;

static void FrameMeterBegin(FrameMeter *m)
{
    const Uint64 now = SDL_GetTicksNS();
    if (m->previous_ns != 0) {
        m->frame_ms = (float)(now - m->previous_ns) / 1000000.0f;
    }
    m->previous_ns = now;
}

static void FrameMeterDraw(SDL_Renderer *r, const FrameMeter *m)
{
    char text[64];
    SDL_snprintf(text, sizeof text, "frame %.2f ms", m->frame_ms);

    SDL_SetRenderDrawColor(r, 255, 255, 255, 255);
    SDL_RenderDebugText(r, 12.0f, 12.0f, text);

    const float width = SDL_min(m->frame_ms * 8.0f, 300.0f);
    const SDL_FRect bar = { 12.0f, 28.0f, width, 6.0f };
    SDL_SetRenderDrawColor(r, 80, 190, 255, 255);
    SDL_RenderFillRect(r, &bar);
}
```

Expected Result

A changing numeric frame time and bar make timing visible without an external profiler.

Experiments That Teach the API

- Cause one intentional slow frame and watch the bar react.
- Track update time and render time separately instead of only total frame time.
- Do not optimize until the overlay proves which phase is expensive.

Debug Habit Change one assumption at a time. When behavior is wrong, log the exact failing SDL call immediately, then verify coordinates, lifetime, target/state, and asset/path assumptions before adding more code.

Make It Yours

- Rename the helper around the vocabulary of your own project instead of preserving tutorial names.
- Keep the function narrow enough that its inputs, outputs, state changes, and owned resources can be explained in one minute.
- After it works, measure one real cost or failure mode before deciding whether another abstraction is useful.

Performance Lens

- Instrumentation must be cheaper than the work it measures. Keep the overlay small, update text at a controlled rate, and avoid per-frame heap allocation.
- Track update, render, present, and total frame time separately when you need actionable evidence; one total number can hide where the stall actually occurs.
- Use a short rolling history or percentile summary for noisy workloads instead of reacting to a single frame that may include OS scheduling or asset I/O.

Failure Modes to Recognize

- Converting nanoseconds with the wrong scale produces plausible-looking but incorrect milliseconds, which can send optimization effort in the wrong direction.
- Formatting large strings or logging every frame can become the bottleneck and distort the measurement you are trying to observe.
- Measuring only CPU submission time may miss blocking in present or GPU work; interpret the overlay as one piece of evidence, not a universal profiler.

CLion Verification

- Set a breakpoint around the timer sample and verify that previous_ns is initialized once and updated exactly once per frame.
- Insert a deliberate delay, confirm the displayed spike, remove it, and verify the overlay returns to its earlier range.

COOKBOOK 06

Audio Trigger with Queue Awareness

Audio streams are the core SDL3 audio path. Open one playback stream, resume its device, push prepared PCM data, and observe how much data remains queued.

Copy-Ready Focus Code

C23 • COPY-READY

```c
static SDL_AudioStream *OpenPlayback(void)
{
    const SDL_AudioSpec src = {
        .format = SDL_AUDIO_F32,
        .channels = 2,
        .freq = 48000
    };

    SDL_AudioStream *stream = SDL_OpenAudioDeviceStream(
        SDL_AUDIO_DEVICE_DEFAULT_PLAYBACK, &src, NULL, NULL);
    if (!stream) {
        SDL_Log("OpenAudioDeviceStream: %s", SDL_GetError());
        return NULL;
    }
    if (!SDL_ResumeAudioStreamDevice(stream)) {
        SDL_Log("ResumeAudioStreamDevice: %s", SDL_GetError());
        SDL_DestroyAudioStream(stream);
        return NULL;
    }
    return stream;
}

static bool PlayChunk(SDL_AudioStream *stream,
                      const float *samples, int bytes)
{
    if (!SDL_PutAudioStreamData(stream, samples, bytes)) return false;
    SDL_Log("queued audio bytes: %d", SDL_GetAudioStreamQueued(stream));
    return true;
}

/* SDL_DestroyAudioStream(stream); closes its opened device too. */
```

Expected Result

The action that triggers sound stays small while the stream owns buffering and device conversion.

Experiments That Teach the API

- Queue the same sound repeatedly and log the queued-byte count.
- Change chunk size without changing the source format.
- Keep decoding and file I/O outside the short trigger function.

Debug Habit Change one assumption at a time. When behavior is wrong, log the exact failing SDL call immediately, then verify coordinates, lifetime, target/state, and asset/path assumptions before adding more code.

Make It Yours

- Rename the helper around the vocabulary of your own project instead of preserving tutorial names.
- Keep the function narrow enough that its inputs, outputs, state changes, and owned resources can be explained in one minute.
- After it works, measure one real cost or failure mode before deciding whether another abstraction is useful.

Performance Lens

- Queue depth is both a stability signal and a latency signal. Enough buffered audio prevents underruns, but excessive queued data makes actions feel delayed.
- Decode and convert reusable sounds before the input-trigger path when possible; a button press should enqueue prepared data, not perform file I/O.
- Keep the stream format stable for a class of sounds so SDL can own device conversion without repeated setup work in the hot interaction path.

Failure Modes to Recognize

- Destroying a stream while game state can still trigger it creates a lifetime bug; make shutdown order explicit and disable producers before cleanup.
- Repeatedly queueing faster than playback consumes data grows latency even though the program appears glitch-free.
- Assuming every PCM buffer already matches the stream contract can produce pitch, channel, or duration errors that look like device problems.

Closing Principle

The strongest SDL3 code is not the code with the most wrappers. It is the code whose contracts are visible: the reader knows what exists, who owns it, which state changes, which coordinate space is active, when a frame is presented, and what must happen when an operation fails. If you can copy a recipe, compile it, vary it, break it deliberately, explain the result, and reshape it into a different application, the book has done its job.