Simple DirectMedia Layer (SDL3)

Gustavo Reis

What is SDL?

The Simple DirectMedia Layer (SDL) is a library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware via modern graphics APIs. SDL is written entirely in C, making it a cross-platform development library that supports Windows, Mac OS X, Linux, iOS, and Android.

Given its portability and ease of use, SDL has become increasingly popular over the years. It is used for video playback software, emulators, games, and popular game engines. Thanks to SDL's portability, most game engines can export the same game for several platforms.

Using SDL3 Library

Initialization and Cleanup

To work with SDL, you need to initialize the library and clean up when finished. Here's the basic structure:

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

int main(int argc, char* argv[]) {
    if (!SDL_Init(SDL_INIT_VIDEO)) {
        SDL_Log("Video initialization error: %s", SDL_GetError());
        return 1;
    }

    // Your code here

    SDL_Quit();
    return 0;
}
```

Key points:

- `SDL_Init()` returns `true` on success, `false` on failure
- `SDL_INIT_VIDEO` initializes the video subsystem
- Always call `SDL_Quit()` before your program ends to free resources
- Use `SDL_GetError()` to get detailed error messages
- `SDL_Log()` is the recommended way to print debug information

Common Initialization Flags

Flag	Description
`SDL_INIT_VIDEO`	Video subsystem for graphics
`SDL_INIT_AUDIO`	Audio subsystem for sound
`SDL_INIT_GAMEPAD`	Gamepad/controller subsystem
`SDL_INIT_JOYSTICK`	Joystick subsystem
`SDL_INIT_HAPTIC`	Haptic (force feedback) subsystem

You can combine flags using the bitwise OR operator:

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_GAMEPAD);
```

Object-Oriented Wrapper (Modern C++)

For better resource management, you can create an RAII wrapper:

```
#include <SDL3/SDL.h>
#include <exception>
#include <string>

class SDLError : public std::exception {
public:
    SDLError() : msg_(SDL_GetError()) {}
    explicit SDLError(const std::string& msg) : msg_(msg) {}

    const char* what() const noexcept override {
        return msg_.c_str();
    }

private:
    std::string msg_;
};

class SDLContext {
public:
    explicit SDLContext(Uint32 flags = 0) {
        if (!SDL_Init(flags)) {
            throw SDLError();
        }
    }

    ~SDLContext() {
        SDL_Quit();
    }
```

```
    // Prevent copying
    SDLContext(const SDLContext&) = delete;
    SDLContext& operator=(const SDLContext&) = delete;
};

// Usage example
int main(int argc, char** argv) {
    try {
        SDLContext sdl(SDL_INIT_VIDEO);

        // Your SDL code here

        return 0;
    }
    catch (const SDLError& err) {
        SDL_Log("SDL Error: %s", err.what());
        return 1;
    }
}
```

Creating a Window

SDL windows are created using the `SDL_CreateWindow()` function:

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

SDL_Window* window = SDL_CreateWindow(
    "Window Title",
    640,              // width
    480,              // height
    SDL_WINDOW_OPENGL // flags
);

if (!window) {
    SDL_Log("Window creation error: %s", SDL_GetError());
    return 1;
}

// Windows are hidden by default, so show it
SDL_ShowWindow(window);
```

Window Creation Parameters

Parameter	Description
`const char* title`	Window title in UTF-8 encoding
`int w`	Window width in screen coordinates

Parameter	Description
`int h`	Window height in screen coordinates
`Uint32 flags`	Window flags (see table below)

Common Window Flags

Flag	Description
`SDL_WINDOW_FULLSCREEN`	Fullscreen window
`SDL_WINDOW_OPENGL`	Window usable with OpenGL context
`SDL_WINDOW_HIDDEN`	Window starts hidden
`SDL_WINDOW_BORDERLESS`	No window decoration
`SDL_WINDOW_RESIZABLE`	Window can be resized
`SDL_WINDOW_MINIMIZED`	Window starts minimized
`SDL_WINDOW_MAXIMIZED`	Window starts maximized
`SDL_WINDOW_HIGH_PIXEL_DENSITY`	High-DPI support

Complete Window Example

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

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

    SDL_Window* window = SDL_CreateWindow(
        "TAPJ",
        640, 480,
        SDL_WINDOW_OPENGL
    );

    if (!window) {
        SDL_Log("Window creation error: %s", SDL_GetError());
        SDL_Quit();
        return 1;
    }

    SDL_ShowWindow(window);
    SDL_Delay(2000);  // Wait 2 seconds

    SDL_DestroyWindow(window);
```

```
    SDL_Quit();

    return 0;
}
```

Working with Textures and Rendering

SDL uses textures for efficient 2D rendering. Textures are stored in GPU memory for fast drawing.

Creating a Renderer

A renderer is responsible for drawing textures and shapes to the window:

```
SDL_Renderer* renderer = SDL_CreateRenderer(window, nullptr);

if (!renderer) {
    SDL_Log("Renderer creation error: %s", SDL_GetError());
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 1;
}
```

The renderer automatically uses hardware acceleration when available.

Loading and Displaying Images

Here's how to load a BMP image and display it:

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

SDL_Texture* LoadTexture(const char* filepath, SDL_Renderer* renderer) {
    SDL_Surface* surface = SDL_LoadBMP(filepath);
    if (!surface) {
        SDL_Log("Failed to load image %s: %s", filepath, SDL_GetError());
        return nullptr;
    }

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

    if (!texture) {
        SDL_Log("Failed to create texture: %s", SDL_GetError());
    }

    return texture;
}

// Usage
int main(int argc, char* argv[]) {
```

```
    if (!SDL_Init(SDL_INIT_VIDEO)) {
        return 1;
    }

    SDL_Window* window = SDL_CreateWindow("Image Display", 640, 480, 0);
    SDL_Renderer* renderer = SDL_CreateRenderer(window, nullptr);
    SDL_ShowWindow(window);

    SDL_Texture* texture = LoadTexture("image.bmp", renderer);

    if (texture) {
        SDL_RenderClear(renderer);
        SDL_RenderTexture(renderer, texture, nullptr, nullptr);
        SDL_RenderPresent(renderer);
        SDL_Delay(2000);

        SDL_DestroyTexture(texture);
    }

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

Key Rendering Functions

Function	Description
`SDL_RenderClear()`	Clears the rendering target
`SDL_RenderTexture()`	Draws a texture to the renderer
`SDL_RenderPresent()`	Updates the screen with rendered content

Game Loop

The game loop is the heart of any game. It continuously processes events, updates game logic, and renders graphics:

```
bool isRunning = true;
SDL_Event event;

while (isRunning) {
    // Event handling
    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_EVENT_QUIT) {
```

```
            isRunning = false;
        }
    }

    // Game logic here

    // Rendering
    SDL_RenderClear(renderer);
    // Draw your textures here
    SDL_RenderPresent(renderer);
}
```

Event Handling

SDL uses an event queue to handle user input. The `SDL_PollEvent()` function retrieves events from the queue.

Keyboard Events

```
while (SDL_PollEvent(&event)) {
    switch (event.type) {
        case SDL_EVENT_QUIT:
            isRunning = false;
            break;

        case SDL_EVENT_KEY_DOWN:
            switch (event.key.keysym.sym) {
                case SDLK_ESCAPE:
                    isRunning = false;
                    break;
                case SDLK_1:
                    currentTexture = texture1;
                    break;
                case SDLK_2:
                    currentTexture = texture2;
                    break;
                case SDLK_SPACE:
                    // Handle spacebar
                    break;
            }
            break;
    }
}
```

Continuous Keyboard State

For smooth movement, check keyboard state directly instead of waiting for events:

```
const bool* keyState = SDL_GetKeyboardState(nullptr);

if (keyState[SDL_SCANCODE_RIGHT]) {
    playerX += moveSpeed * deltaTime;
}
if (keyState[SDL_SCANCODE_LEFT]) {
    playerX -= moveSpeed * deltaTime;
}
if (keyState[SDL_SCANCODE_UP]) {
    playerY -= moveSpeed * deltaTime;
}
if (keyState[SDL_SCANCODE_DOWN]) {
    playerY += moveSpeed * deltaTime;
}
```

Mouse Events

Handle mouse movement and clicks:

```
while (SDL_PollEvent(&event)) {
    if (event.type == SDL_EVENT_MOUSE_MOTION) {
        float mouseX = event.motion.x;
        float mouseY = event.motion.y;
        SDL_Log("Mouse at: %.0f, %.0f", mouseX, mouseY);
    }
    else if (event.type == SDL_EVENT_MOUSE_BUTTON_DOWN) {
        if (event.button.button == SDL_BUTTON_LEFT) {
            SDL_Log("Left click at: %f, %f",
                    event.button.x, event.button.y);
        }
        else if (event.button.button == SDL_BUTTON_RIGHT) {
            SDL_Log("Right click");
        }
    }
}
```

Gamepad Support

SDL has excellent gamepad support with automatic controller mapping:

```
// Initialize gamepad subsystem
if (!SDL_Init(SDL_INIT_GAMEPAD)) {
    SDL_Log("Failed to initialize gamepad: %s", SDL_GetError());
    return 1;
}

// Get available gamepads
```

```
int numGamepads = 0;
SDL_JoystickID* gamepads = SDL_GetGamepads(&numGamepads);

if (numGamepads > 0) {
    SDL_Gamepad* gamepad = SDL_OpenGamepad(gamepads[0]);

    if (gamepad) {
        SDL_Log("Gamepad connected: %s", SDL_GetGamepadName(gamepad));
    }
}

SDL_free(gamepads);
```

Gamepad Events

Handle button presses and analog stick movement:

```
while (SDL_PollEvent(&event)) {
    if (event.type == SDL_EVENT_GAMEPAD_BUTTON_DOWN) {
        switch (event.gbutton.button) {
            case SDL_GAMEPAD_BUTTON_SOUTH:  // A (Xbox) / Cross (PS)
                SDL_Log("South button pressed");
                break;
            case SDL_GAMEPAD_BUTTON_EAST:   // B (Xbox) / Circle (PS)
                SDL_Log("East button pressed");
                break;
            case SDL_GAMEPAD_BUTTON_START:
                SDL_Log("Start button pressed");
                break;
        }
    }
    else if (event.type == SDL_EVENT_GAMEPAD_AXIS_MOTION) {
        if (event.gaxis.axis == SDL_GAMEPAD_AXIS_LEFTX) {
            // Normalize axis value from -32768..32767 to -1.0..1.0
            float axisValue = event.gaxis.value / 32767.0f;
            playerX += axisValue * moveSpeed * deltaTime;
        }
        else if (event.gaxis.axis == SDL_GAMEPAD_AXIS_LEFTY) {
            float axisValue = event.gaxis.value / 32767.0f;
            playerY += axisValue * moveSpeed * deltaTime;
        }
    }
}
```

Common Gamepad Buttons

Button	Xbox	PlayStation
`SDL_GAMEPAD_BUTTON_SOUTH`	A	Cross
`SDL_GAMEPAD_BUTTON_EAST`	B	Circle
`SDL_GAMEPAD_BUTTON_WEST`	X	Square
`SDL_GAMEPAD_BUTTON_NORTH`	Y	Triangle
`SDL_GAMEPAD_BUTTON_START`	Menu	Options
`SDL_GAMEPAD_BUTTON_GUIDE`	Xbox	PS Button

Timing and Delta Time

Frame-rate independent movement requires delta time (time elapsed since last frame):

```
Uint64 previousTime = SDL_GetTicks();
float deltaTime = 0.0f;

while (isRunning) {
    Uint64 currentTime = SDL_GetTicks();
    deltaTime = (currentTime - previousTime) / 1000.0f;  // Convert to seconds
    previousTime = currentTime;

    // Event handling
    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_EVENT_QUIT) {
            isRunning = false;
        }
    }

    // Update game logic with deltaTime
    playerX += velocity * deltaTime;

    // Rendering
    SDL_RenderClear(renderer);
    // Render objects
    SDL_RenderPresent(renderer);
}
```

Note: `SDL_GetTicks()` returns milliseconds as `Uint64`. Divide by 1000.0 to get seconds.

Why Delta Time Matters

Without delta time, movement speed depends on frame rate: - 60 FPS: object moves at normal speed - 30 FPS: object moves at half speed - 120 FPS: object moves at double speed

With delta time, movement is consistent regardless of frame rate.

Sprite Animation

A sprite sheet contains multiple frames of animation in a single image. Here's how to animate a sprite:

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

struct SpriteSheet {
    SDL_Texture* texture = nullptr;
    int frameWidth = 0;
    int frameHeight = 0;
    int numFramesX = 0;
    int numFramesY = 0;
    int currentFrame = 0;
    float animationTimer = 0.0f;
    float frameDuration = 0.1f; // seconds per frame
};

void UpdateAnimation(SpriteSheet& sprite, float deltaTime) {
    sprite.animationTimer += deltaTime;

    if (sprite.animationTimer >= sprite.frameDuration) {
        sprite.animationTimer = 0.0f;
        sprite.currentFrame++;

        int totalFrames = sprite.numFramesX * sprite.numFramesY;
        if (sprite.currentFrame >= totalFrames) {
            sprite.currentFrame = 0;
        }
    }
}

SDL_FRect GetFrameRect(const SpriteSheet& sprite) {
    int frameX = sprite.currentFrame % sprite.numFramesX;
    int frameY = sprite.currentFrame / sprite.numFramesX;

    return SDL_FRect{
        static_cast<float>(frameX * sprite.frameWidth),
        static_cast<float>(frameY * sprite.frameHeight),
        static_cast<float>(sprite.frameWidth),
        static_cast<float>(sprite.frameHeight)
    };
}
```

Complete Animation Example

```
int main(int argc, char* argv[]) {
    if (!SDL_Init(SDL_INIT_VIDEO)) {
```

```
    return 1;
}

SDL_Window* window = SDL_CreateWindow("Sprite Animation", 640, 480, 0);
SDL_Renderer* renderer = SDL_CreateRenderer(window, nullptr);
SDL_ShowWindow(window);

// Load sprite sheet (256x64 pixels with 8x2 grid)
SpriteSheet drone;
SDL_Surface* surface = SDL_LoadBMP("drone.bmp");
if (surface) {
    // Set magenta (255, 0, 255) as transparent
    SDL_SetSurfaceColorKey(surface, true,
                           SDL_MapRGB(surface->format, 255, 0, 255));
    drone.texture = SDL_CreateTextureFromSurface(renderer, surface);

    int textureWidth, textureHeight;
    SDL_GetTextureSize(drone.texture, &textureWidth, &textureHeight);

    // Calculate frame dimensions (8x2 grid)
    drone.numFramesX = 8;
    drone.numFramesY = 2;
    drone.frameWidth = textureWidth / drone.numFramesX;
    drone.frameHeight = textureHeight / drone.numFramesY;

    SDL_DestroySurface(surface);
}

SDL_FRect playerPos = {100.0f, 100.0f, 64.0f, 64.0f};
float moveSpeed = 200.0f;

Uint64 previousTime = SDL_GetTicks();
bool isRunning = true;
SDL_Event event;

while (isRunning) {
    Uint64 currentTime = SDL_GetTicks();
    float deltaTime = (currentTime - previousTime) / 1000.0f;
    previousTime = currentTime;

    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_EVENT_QUIT) {
            isRunning = false;
        }
    }

    // Handle input
    const bool* keys = SDL_GetKeyboardState(nullptr);
```

```
        if (keys[SDL_SCANCODE_RIGHT]) {
            playerPos.x += moveSpeed * deltaTime;
        }
        if (keys[SDL_SCANCODE_LEFT]) {
            playerPos.x -= moveSpeed * deltaTime;
        }

        // Update animation
        UpdateAnimation(drone, deltaTime);

        // Render
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
        SDL_RenderClear(renderer);

        SDL_FRect srcRect = GetFrameRect(drone);
        SDL_RenderTexture(renderer, drone.texture, &srcRect, &playerPos);

        SDL_RenderPresent(renderer);
    }

    SDL_DestroyTexture(drone.texture);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

Color Keying (Transparency)

BMP files don't support transparency, so you must specify which color should be transparent:

```
SDL_Texture* LoadTexture(const char* filepath, SDL_Renderer* renderer) {
    SDL_Surface* surface = SDL_LoadBMP(filepath);
    if (!surface) {
        SDL_Log("Failed to load image: %s", SDL_GetError());
        return nullptr;
    }

    // Set magenta (255, 0, 255) as transparent
    SDL_SetSurfaceColorKey(surface, true,
                           SDL_MapRGB(surface->format, 255, 0, 255));

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

```
    return texture;
}
```

Complete Game Example

Here's a complete game application with proper structure:

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

class Game {
public:
    Game() : running_(false), window_(nullptr), renderer_(nullptr) {}

    bool Initialize(const char* title, int width, int height) {
        if (!SDL_Init(SDL_INIT_VIDEO | SDL_INIT_GAMEPAD)) {
            SDL_Log("SDL initialization failed: %s", SDL_GetError());
            return false;
        }

        window_ = SDL_CreateWindow(title, width, height, 0);
        if (!window_) {
            SDL_Log("Window creation failed: %s", SDL_GetError());
            return false;
        }

        renderer_ = SDL_CreateRenderer(window_, nullptr);
        if (!renderer_) {
            SDL_Log("Renderer creation failed: %s", SDL_GetError());
            return false;
        }

        SDL_ShowWindow(window_);
        return true;
    }

    void Run() {
        running_ = true;
        Uint64 previousTime = SDL_GetTicks();

        while (running_) {
            Uint64 currentTime = SDL_GetTicks();
            float deltaTime = (currentTime - previousTime) / 1000.0f;
            previousTime = currentTime;

            HandleEvents();
            Update(deltaTime);
            Render();
```

```
        }
    }

    void Shutdown() {
        if (renderer_) SDL_DestroyRenderer(renderer_);
        if (window_) SDL_DestroyWindow(window_);
        SDL_Quit();
    }

private:
    void HandleEvents() {
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_EVENT_QUIT) {
                running_ = false;
            }
            else if (event.type == SDL_EVENT_KEY_DOWN) {
                if (event.key.keysym.sym == SDLK_ESCAPE) {
                    running_ = false;
                }
            }
        }
    }

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

    void Render() {
        SDL_SetRenderDrawColor(renderer_, 0, 0, 0, 255);
        SDL_RenderClear(renderer_);

        // Render game objects here

        SDL_RenderPresent(renderer_);
    }

    SDL_Window* window_;
    SDL_Renderer* renderer_;
    bool running_;
};

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

    if (!game.Initialize("SDL3 Game", 640, 480)) {
        return 1;
    }
```

```
    game.Run();
    game.Shutdown();

    return 0;
}
```

Best Practices

1. **Always check return values** - Most SDL functions return `false` or `nullptr` on failure
2. **Use delta time** - Ensure consistent behavior across different frame rates
3. **Free resources** - Always destroy textures, renderers, and windows
4. **Use `SDL_Log()`** - Better than `printf` for SDL debugging
5. **Handle events every frame** - Empty the event queue to prevent lag
6. **Use `SDL_FRect`** - Provides floating-point precision for smooth movement
7. **Organize your code** - Use classes or functions to structure your game

Common Pitfalls

- Forgetting to call `SDL_ShowWindow()` (windows are hidden by default)
- Not checking if textures loaded successfully
- Using absolute movement instead of delta-time-based movement
- Not destroying resources, causing memory leaks
- Forgetting to call `SDL_RenderPresent()` after drawing

Resources

- Official SDL3 Documentation
- SDL3 GitHub Repository
- SDL3 API Reference