Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This slideset and associated source code may not be distributed without prior written notice
- This slideset may not be mined using AI or algorithms and otherwise generating derivative products without permission.

Please do not redistribute slides/source without prior written permission.

Graphics Programming with SDL 3

MIKE SHAH

Graphics Programming with SDL 3

MIKE SHAH

Web: mshah.io
YouTube www.youtube.com/c/MikeShah
Social: mikeshah.bsky.social
Courses: courses.mshah.io
Talks: http://tinyurl.com/mike-talks

60 minutes | Audience: Beginner
1:30pm - 2:30pm Fri, Sept. 19, 2025

Abstract (Which you already read :))

Talk Abstract: The C++ programming language does not have a standard graphics library, However, there exists many popular graphics frameworks for cross-platform graphics. In this talk, I will provide an introduction to the Simple Directmedia Layer (SDL) library, which has at the start of 2025 released version 3. This library for several decades has been a standard in the games and graphics industry. Throughout this talk, I will show how to get started with the library, some more advanced examples (including compiling graphics applications to web), and then talk about what a standard graphics library could look like in C++, or if it is even necessary. I will also talk about the 3D GPU library in SDL3. Attendees will leave this talk ready to build multimedia / game applications and with an understanding on if SDL3 is the right tool for them.

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

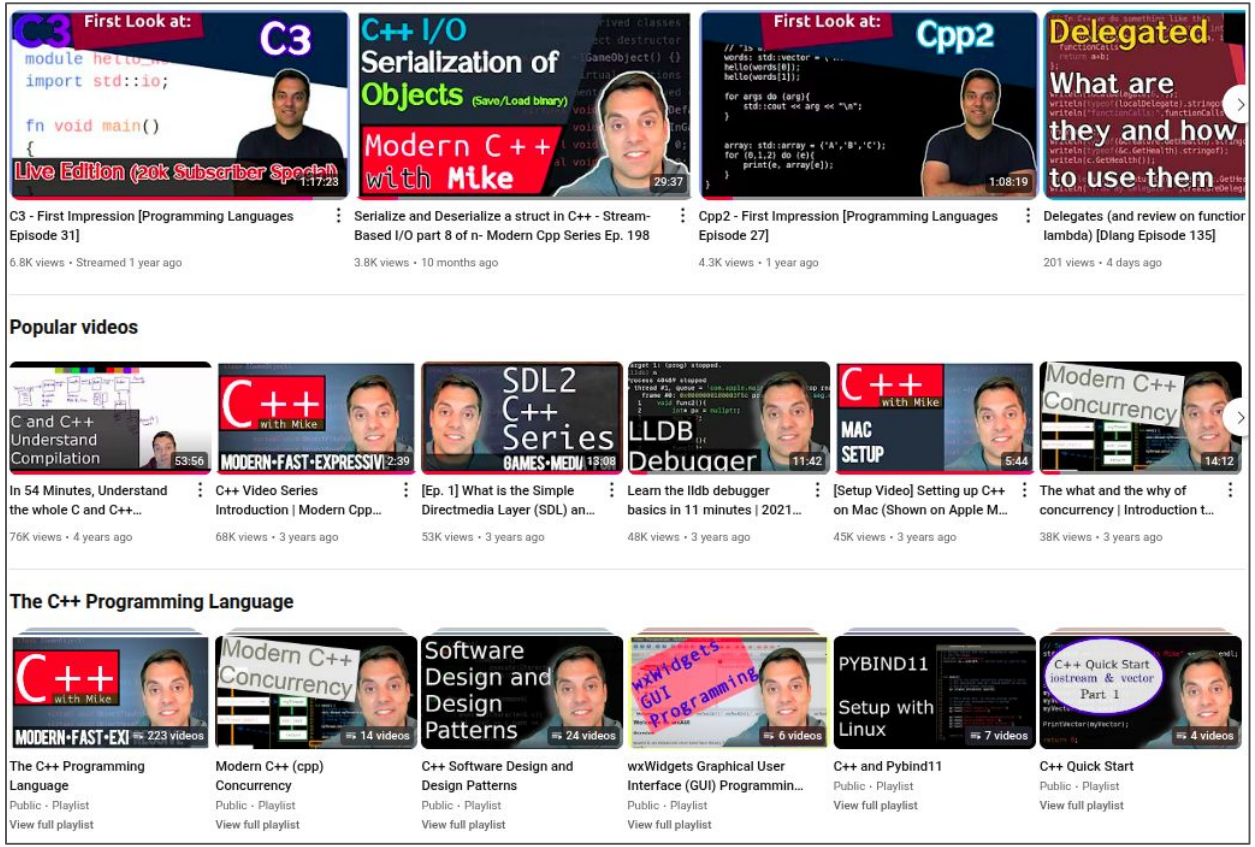
Your Tour Guide for Today

Mike Shah

- **My role at Cppcon:**
  - **Co-Chair of Back to Basics Track (**2022-2025) and **Software-Design Track** (2021-2025)
- **Current Job:** Teaching Faculty at **Yale University**
  - **Teach/Research**: computer systems, graphics, geometry, game engines, software engineering, and computer science education.
- **Available for:**
  - **Technical Training/Talks:** Occasionally
    - Please reach out, and I'll let you know if I have expertise that might add value for short term arrangements
- **Fun**:
  - Enjoy travel, NBA basketball, Olympics, running/weights, video games, music, DuoLingo (French mostly), and pizza making

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Find my programming content on YouTube
https://www.youtube.com/c/mikeshah



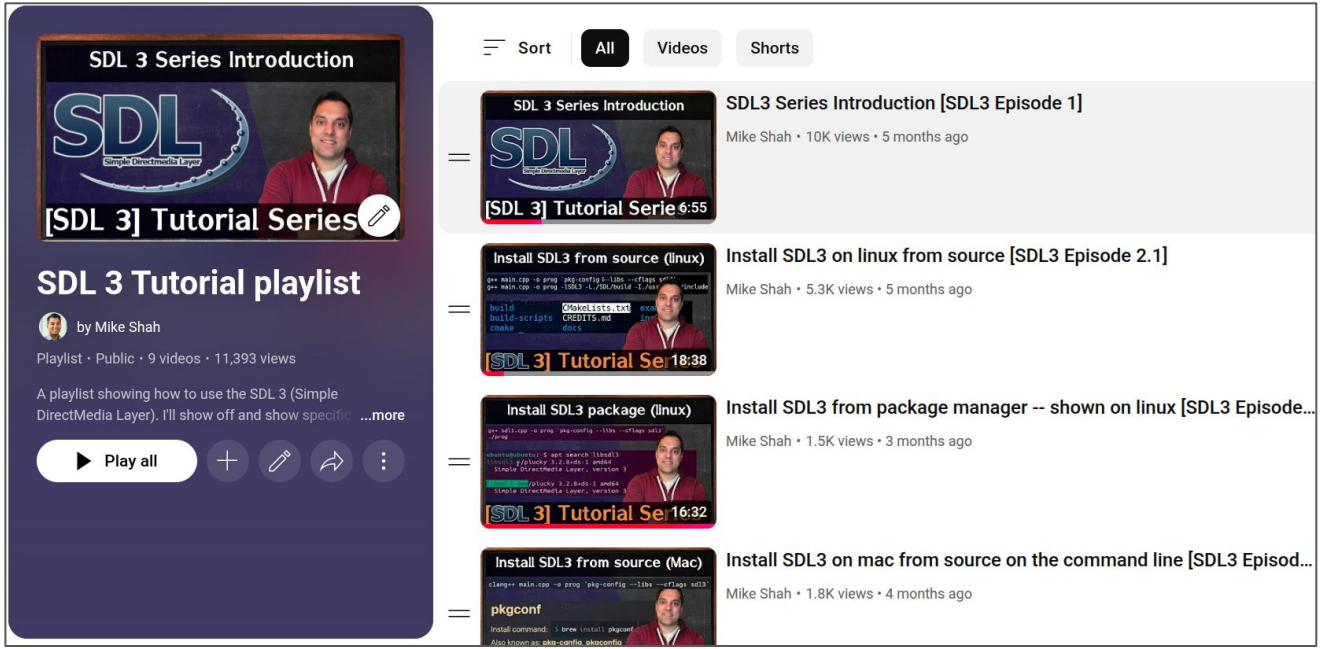
Web
www.mshah.io

YouTube
https://www.youtube.com/c/MikeShah

Non-Academic Courses
courses.mshah.io

Conference Talks
http://tinyurl.com/mike-talks

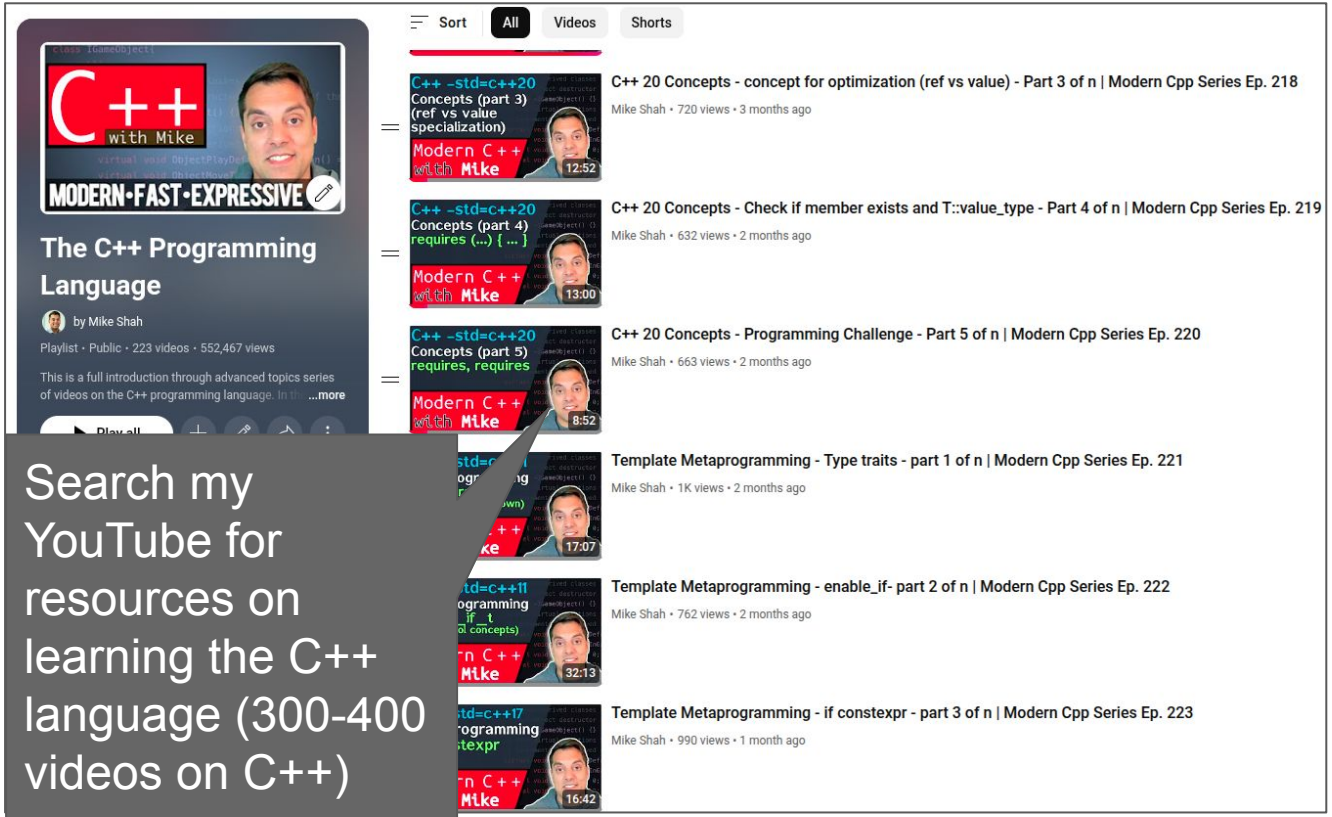
SDL3 Mike Shah



I have many SDL3 videos on YouTube!
https://www.youtube.com/playlist?list=PLvv0ScY6vfd-RZSmGbLkZvkgec6lJ0BfX

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

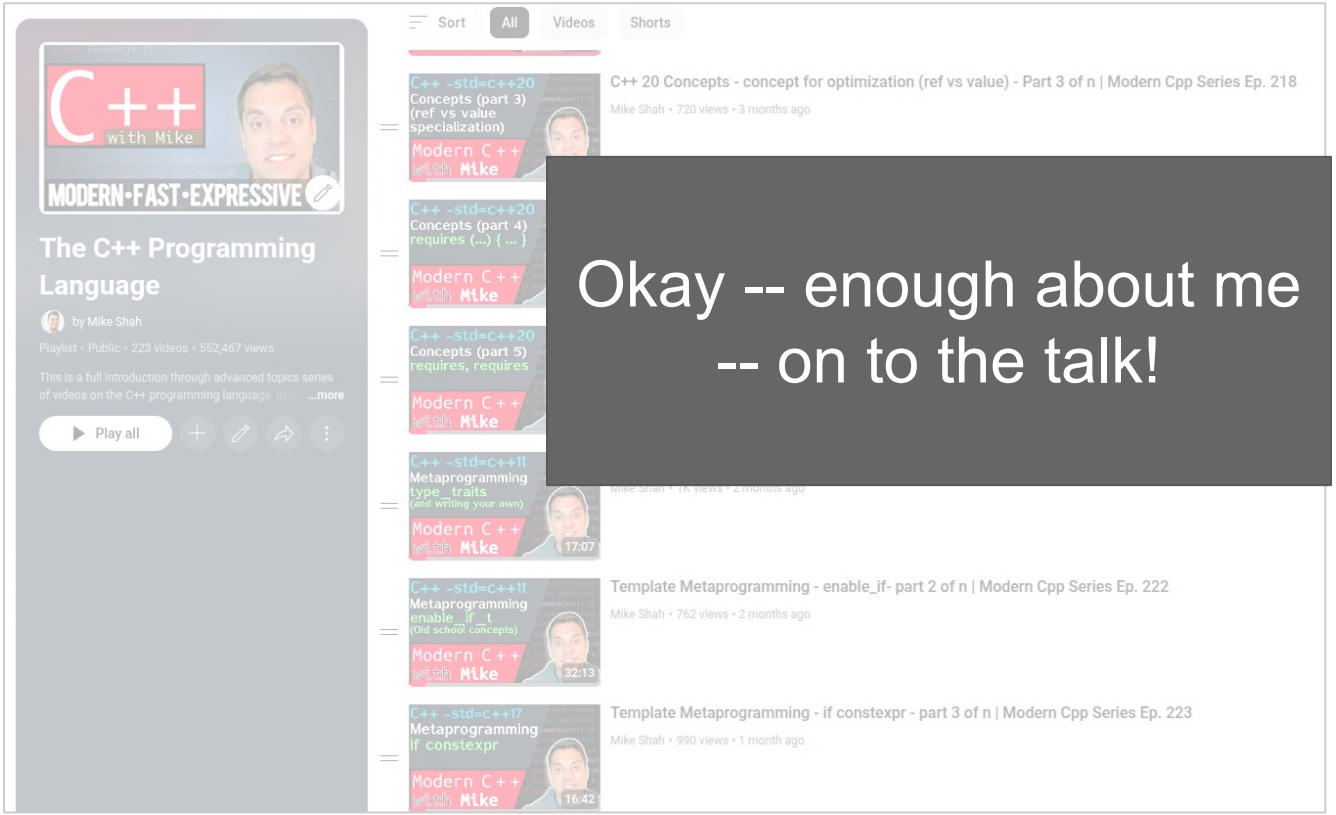
Find my programming content on YouTube
https://www.youtube.com/c/mikeshah



Search my YouTube for resources on learning the C++ language (300-400 videos on C++)

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Find my programming content on YouTube
https://www.youtube.com/c/mikeshah

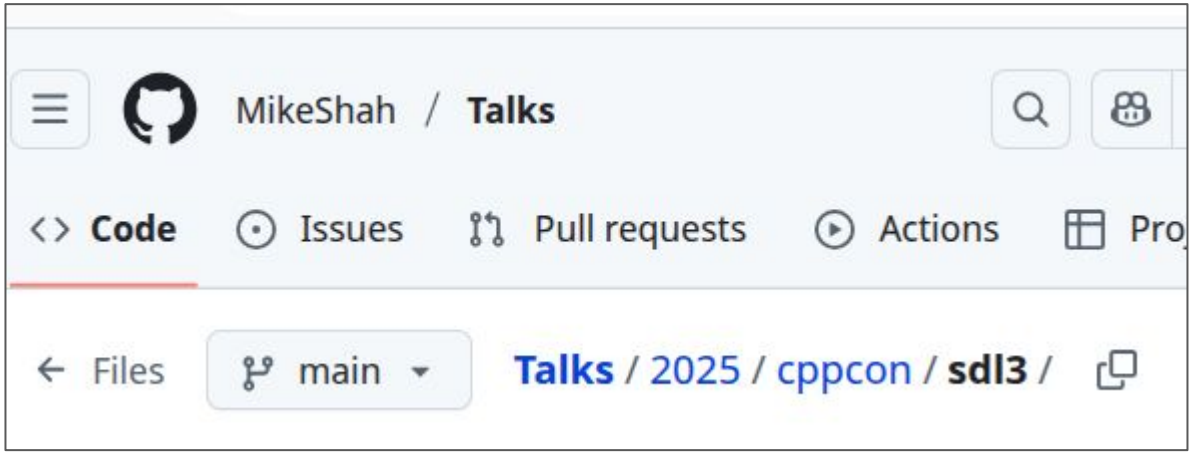


Okay -- enough about me -- on to the talk!

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Source Code

- Source code is available here
- https://github.com/MikeShah/Talks/tree/main/2025/cppcon/sdl3
  - Note: These slides will be posted to my site, where you can find this link again too.

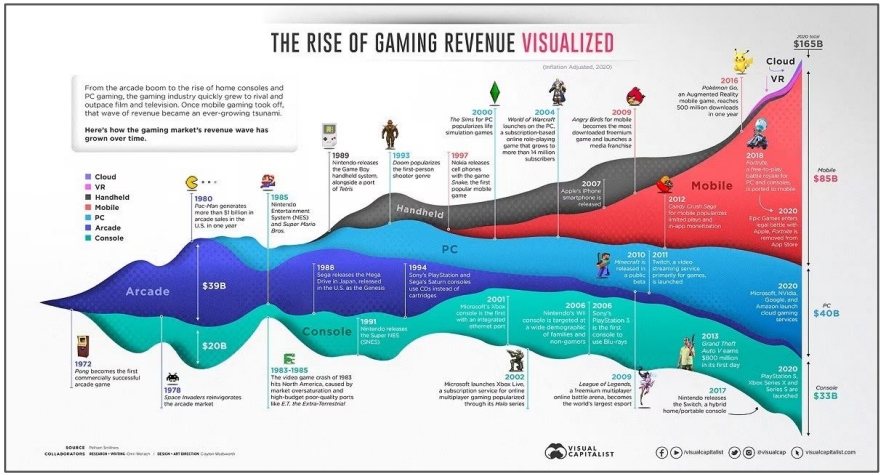


Question to Audience: How many of you got your start programming because you wanted to make games?

A Brief Aside:
The Business of Video Games

The Game Industry (1/2)

- The game industry grew from 2017 and 2021 from $131 billion to $211 billion
  - This is from the revenues of major AAA* game studios
- 2023 to present day has been a particularly tough time on the industry regarding turnover
  - Yet, the industry remains projected to continue growing
  - https://www.bcg.com/press/12december2024-future-of-global-gaming-industry

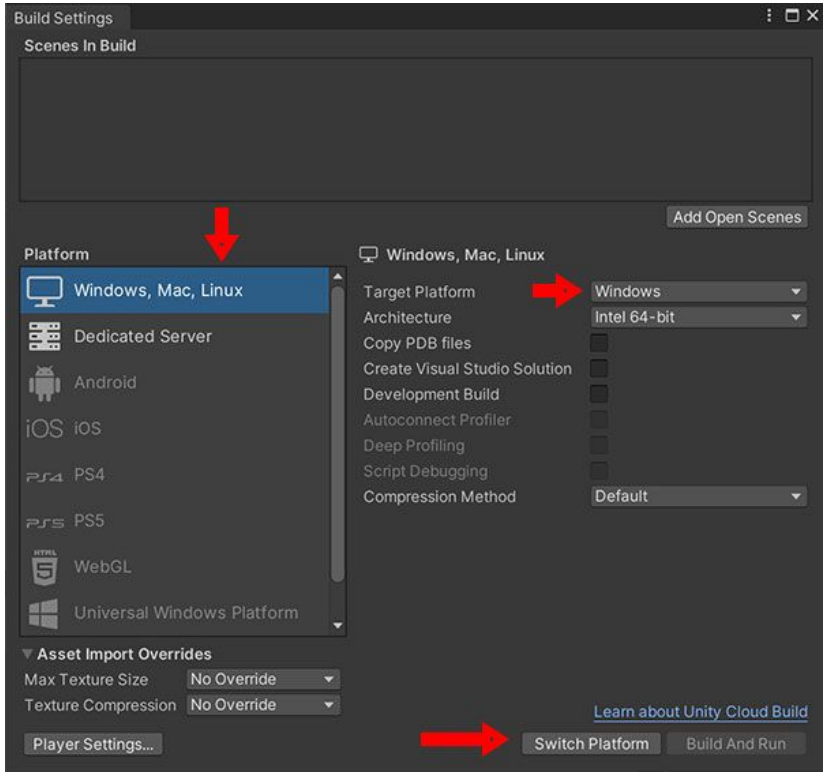


2020. https://www.weforum.org/stories/2020/11/gaming-games-consels-xbox-play-station-fun/

AAA* or 'triple A' is the general term used for large budget, and big (100s of employee) game companies. Often these companies have large budgets and can produce pretty incredible games. Think 'God of War', 'Madden Football', 'Diablo', etc. as example titles.

The Game Industry (2/2)

- With the business side of games in mind -- the '**export game**' dialog box is quite important!
- An important part of games reaching a large audience (and helping generate revenue), is the ability to run on multiple platforms/devices.
  - Our goal is to get games to as many people as possible!
  - If you're a big or small game developer, one of the competitive advantages you can give yourself, is to deploy to multiple platforms



Game Engines like Unreal Engine and GoDot have similar screens to export to a variety of platforms.

How do programmers support multiple platforms? (1/2)

- We could leverage polymorphism for specific instances -- or perhaps that is too costly for performance.
- So we could instead define totally separate projects for each platform… or utilize the preprocessor
  - In C++ we have `#if defined` statements that we often write for each platform.
  - Typically these `#if/#elif#else` preprocessor commands creep in on a ‘per-function’ basis

Source from: /SDL/src/cpuinfo/SDL_cpuinfo.c

How do programmers support multiple platforms? (2/2)

- Note: We might even leverage our build system to implement specific platform functionality
  - Either way -- this is potentially a lot of work.
  - And the parts of code that needs to run on multiple platforms (e.g. the window, input, display) probably do not have as much to do with the actual game we are programming!
- **If only there was a well tested cross-platform library that handled all this for us...** 🤔
  - **(next slide!)**

```
1  // @file: cross_platform_example/example.cpp
2
3  #ifdef WINDOWS
4      #include "Windows_Input_API.h"
5  #elif LINUX
6      #include "Linux_Input_API.h"
7  #else
8      static_assert(0,"Platform not supported");
9  #endif
10
11 int main(){
12     return 0;
13 }
14
```

Note: It will be a good idea to have some unit tests to make sure you have parity (i.e. each function is implemented on each platform the user expects)

Note: There are some nice tricks on handling complicated #ifdef I learned from the book Beautiful C++

Some Exciting News:

There are libraries that solve this exact problem of supporting multiple target platforms

Today's Goal:

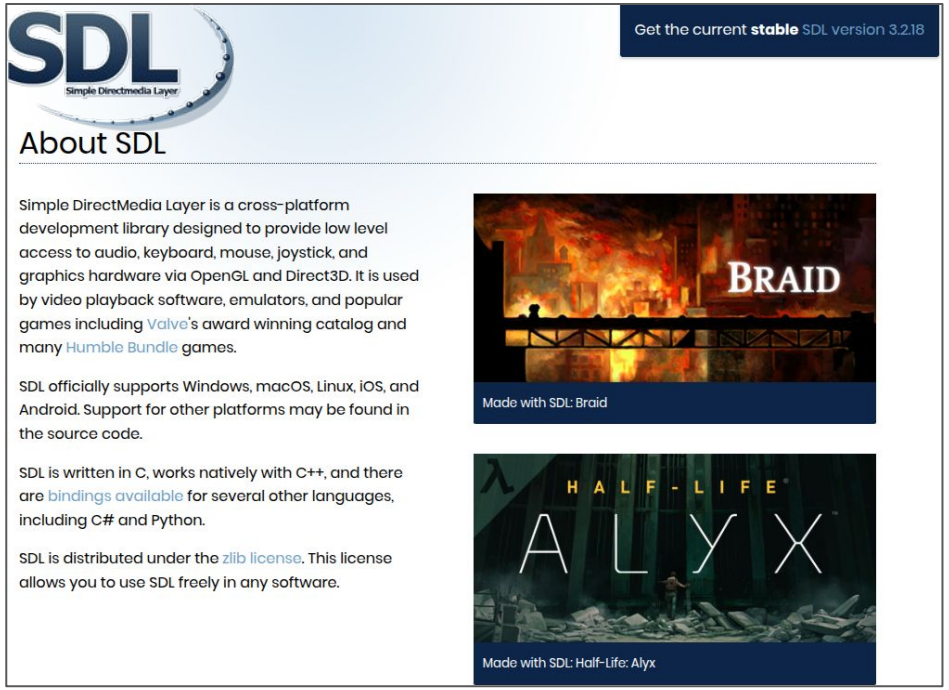
Show you how to use the SDL 3 Graphics Library with several examples

Simple **D**irectmedia **L**ayer (SDL)

A multiplatform library for games and media applications written in C

Simple Directmedia Layer (SDL)

- SDL is a C library for handling **multiplatform** support for:
  - windowing, audio, input, graphics, I/O, displays, camera, and more!
  - SDL can be statically or dynamically linked
- SDL supports:
    - Windows, Linux, Mac, *BSD
    - Android, iOS, tvOS,
    - PlayStation, Nintendo Switch
    - Emscripten (for Web)
    - Older generation consoles: (Nintendo 3DS, PS2, PS Vita, etc.)
    - **And many more platforms!**



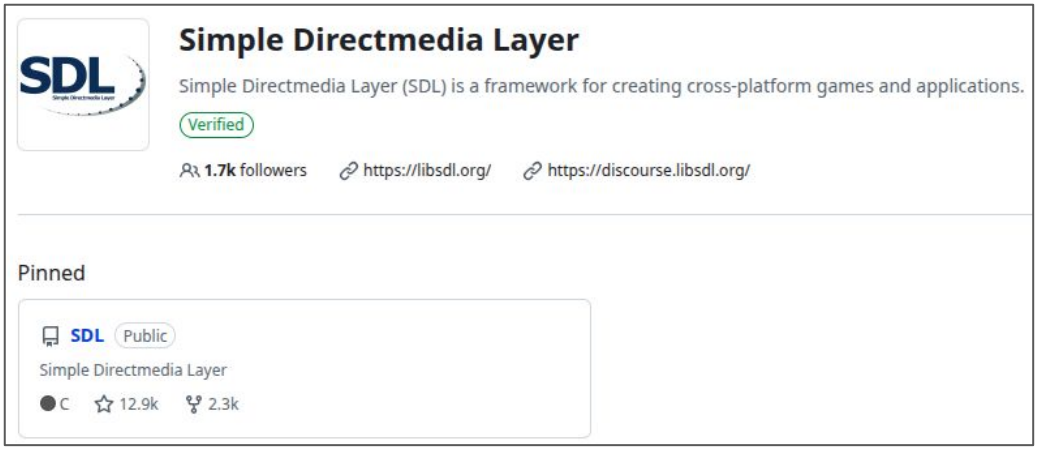
SDL History (27+ Years)

- The first version of SDL was born out of need from someone developing a 68k Macintosh emulator to otherwise support multiple platforms.
  - https://www.macintoshrepository.org/47787-executor
- Sam Lantinga (the original/current SDL author) figured it would be nice to have all the common code for:
  - window creation, device input, audio, some graphics, etc. all in one place
- Ryan Gordon is one of the other main authors of the library today

More history:
Wiki: https://en.wikipedia.org/wiki/Simple_DirectMedia_Layer
GDC Talk:
https://gdcvault.com/play/1029269/Open-Source-Game-Development-Summit)
Reddit AMA:
https://www.reddit.com/r/gamedev/comments/1bro6ni/we_are_the_developers_of_sdl_ask_us_anything/

Simple Directmedia Layer (SDL) is Open Source

- SDL is open source so you can see all of the implementation details and add to the library as needed
- Because SDL is ‘C’ based there exist bindings to many programming languages
- **SDL works trivially with C++**
  - Bindings are otherwise available or otherwise trivial for languages that can interface with C (e.g. Dlang)
- So effort put into learning SDL will benefit you long term -- you can use it in C, C++, D, Go, Rust, Pascal, Python, Swift, etc.



https://github.com/libsdl-org

Note: I have even had current students in my class land pull requests and bug fixes!

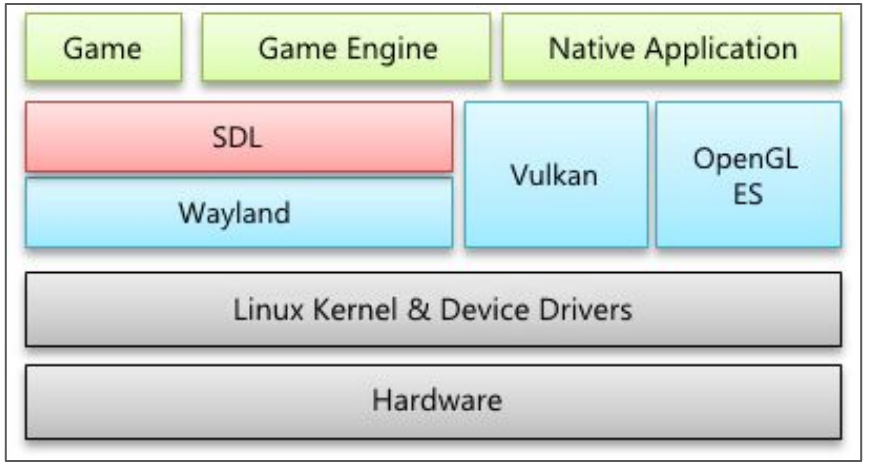
Simple Directmedia Layer (SDL) is Industry Proven

- SDL is an industry proven (27+ years) library.
  - Thousands of games, game engines, and multimedia applications have and will continue to use the SDL Library
  - This includes both large and small game studios
  - And of course, many gamers play games built on top of SDL!
- SDL has largely had a very stable set of standardized functions developed to assist in:
  - Gaming
  - Graphics Tools
  - Other Multimedia applications (Sound/Video)

https://media.steampowered.com/apps/steamdevdays/slides/sdl2.pdf

SDL is not a full game/graphics engine

- SDL provides an abstraction layer that sits between your game and the operating system
  - It's simply a library serving as a 'game framework' to build on
- Note:
  - SDL does provide a lightweight 2D graphics API for drawing graphics
  - That itself has been enough to build complete games in -- we will talk about that shortly!



https://docs.tizen.org/application/native/guides/graphics/sdl/

The term 'platform independent' means that you can run on more than one platform with the same code (i.e. linux, windows, mac, etc.)

SDL Abstraction Layers

- Through SDL's abstraction layer, SDL provides a 'platform independent*' way to:
  - Setup a window in your operating system
  - Render graphics
    - Either software or hardware based
    - For hardware you can render with:
      - OpenGL, Vulkan, Direct3D, Metal, etc.
  - Handle audio and audio devices
  - Threading
  - Handle File I/O
  - Handle a variety of input devices
    - Pen devices, touch display, keyboard, game controller, etc.
  - and more!

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

Video

View information and functions related to...	View the header
Display and Window Management	SDL_video.h
2D Accelerated Rendering	SDL_render.h
Pixel Formats and Conversion Routines	SDL_pixels.h
Blend modes	SDL_blendmode.h
Rectangle Functions	SDL_rect.h
Surface Creation and Simple Drawing	SDL_surface.h
Clipboard Handling	SDL_clipboard.h
Vulkan Support	SDL_vulkan.h
Metal Support	SDL_metal.h
Camera Support	SDL_camera.h

Input Events

View information and functions related to...	View the header
Event Handling	SDL_events.h
Keyboard Support	SDL_keyboard.h
Keyboard Keycodes	SDL_keycode.h
Keyboard Scancodes	SDL_scancode.h
Mouse Support	SDL_mouse.h
Joystick Support	SDL_joystick.h
Gamepad Support	SDL_gamepad.h
Touch Support	SDL_touch.h
Pen Support	SDL_pen.h
Sensors	SDL_sensor.h
HIDAPI	SDL_hidapi.h

Force Feedback ("Haptic")

View information and functions related to...	View the header
Force Feedback Support	SDL_haptic.h

Audio

Related Frameworks

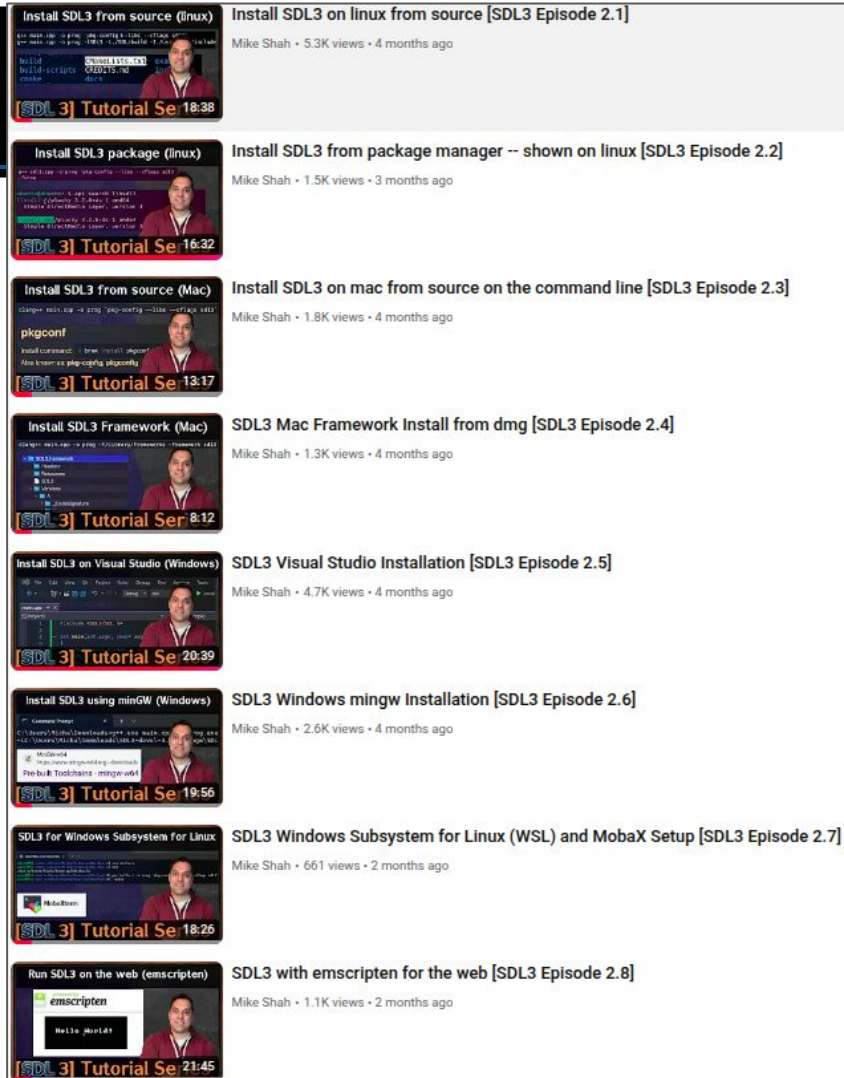
- There exist other cross-platform frameworks
  - SFML (Modern C++ Library)
    - https://www.sfml-dev.org/
  - GLFW (Cross-Platform Windowing/Input framework for use with OpenGL, OpenGL ES, and Vulkan)
    - https://www.glfw.org/
  - QT
    - https://www.qt.io/
  - And many more (wxWidgets, gtk, Cairo, etc.)

How to Get Started with SDL

Installation and Setup

Install/Setup SDL (1/2)

- SDL 3 is a library, so that means we need to either download or build a static or dynamic version of the library
- The compilation process is well documented by the SDL team and also by video from me on many platforms
  - SDL Wiki: https://github.com/libsdl-org/SDL/blob/main/INSTALL.md
  - My C++ SDL3 Installation Videos: https://www.youtube.com/playlist?list=PLvv0ScY6vfd-RZSmGbLkZvkgec6lJ0BfX



Install/Setup SDL (2/2)

- You can also explore build systems and package managers:
  - Conan
    - https://conan.io/center/recipes/sdl
  - vcpkg
    - https://vcpkg.io/en/package/sdl3
  - cmake
    - https://github.com/libsdl-org/SDL/blob/main/docs/INTRO-cmake.md

Install SDL3 on linux from source [SDL3 Episode 2.1]
Mike Shah • 5.3K views • 4 months ago

Install SDL3 from package manager -- shown on linux [SDL3 Episode 2.2]
Mike Shah • 1.5K views • 3 months ago

Install SDL3 on mac from source on the command line [SDL3 Episode 2.3]
Mike Shah • 1.8K views • 4 months ago

SDL3 Mac Framework Install from dmg [SDL3 Episode 2.4]
Mike Shah • 1.3K views • 4 months ago

SDL3 Visual Studio Installation [SDL3 Episode 2.5]
Mike Shah • 4.7K views • 4 months ago

SDL3 Windows mingw Installation [SDL3 Episode 2.6]
Mike Shah • 2.6K views • 4 months ago

SDL3 Windows Subsystem for Linux (WSL) and MobaX Setup [SDL3 Episode 2.7]
Mike Shah • 661 views • 2 months ago

SDL3 with emscripten for the web [SDL3 Episode 2.8]
Mike Shah • 1.1K views • 2 months ago

A “Hello World” SDL Program

In the **C++** Programming Language

SDL C++ Program

- This is it -- this is the basic program that will initialize SDL and setup and display a window to your screen.
  - Pretty simple, huh?
- The impressive part is that this will work across basically every operating system you would want to develop a game or media application on.

```
1  // @file: 00_commented.cpp
2  +--  7 lines: // linux:    ------------------------------
9  #include <SDL3/SDL.h>
10
11 int main(int argc, char* argv[]){
12
13     // Initialize SDL with the video subsystem.
14     // If it returns less than false then an
15     // error code will be received.
16     SDL_Init(SDL_INIT_VIDEO);
17
18     // Create a window data type
19     // This pointer will point to the
20     // window that is allocated from SDL_CreateWindow
21     SDL_Window* window;
22     // Request a window to be created for our platform
23     // The parameters are for the title and the width
24     // and height of the window.
25     window = SDL_CreateWindow("Hello C++ North", 320, 240,
26                               SDL_WINDOW_RESIZABLE);
27
28     // We add a delay in order to see that our window
29     // has successfully popped up.
30     SDL_Delay(3000);
31
32     // We destroy our window. We are passing in the pointer
33     // that points to the memory allocated by the
34     // 'SDL_CreateWindow' function. Remember, this is
35     // a 'C-style' API, we don't have destructors.
36     SDL_DestroyWindow(window);
37
38     // We safely uninitialize SDL2, that is, we are
39     // taking down the subsystems here before we exit
40     // our program.
41     SDL_Quit();
42
43     return 0;
44 }
```

Compiling an SDL Program

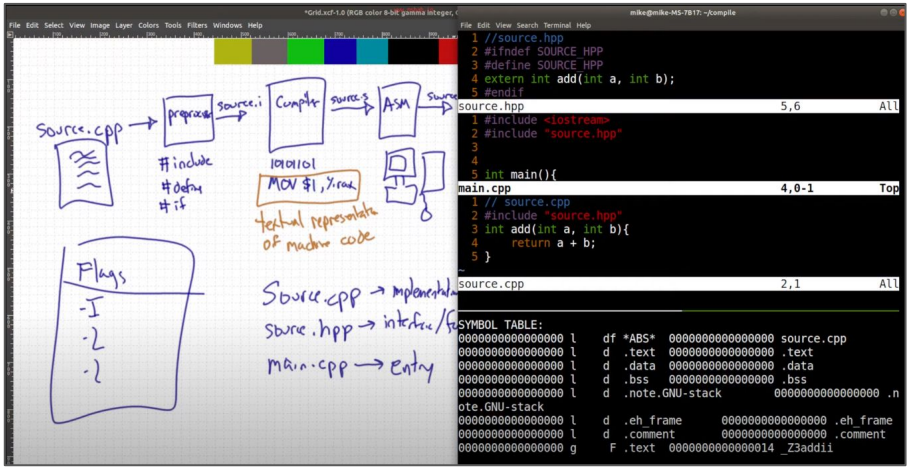
- Linux C++ code compilation
  - `g++ main.cpp -o prog -lSDL3`
- Or using pkg-config
  - `g++ main.cpp -o prog `pkg-config --libs --cflags sdl3``
- Or cmake, etc.

```
1  // @file: 00_commented.cpp
2  +--  7 lines: // linux:    ------------------------------
9  #include <SDL3/SDL.h>
10
11 int main(int argc, char* argv[]){
12
13     // Initialize SDL with the video subsystem.
14     // If it returns less than false then an
15     // error code will be received.
16     SDL_Init(SDL_INIT_VIDEO);
17
18     // Create a window data type
19     // This pointer will point to the
20     // window that is allocated from SDL_CreateWindow
21     SDL_Window* window;
22     // Request a window to be created for our platform
23     // The parameters are for the title and the width
24     // and height of the window.
25     window = SDL_CreateWindow("Hello C++ North", 320, 240,
26                               SDL_WINDOW_RESIZABLE);
27
28     // We add a delay in order to see that our window
                successfully popped up.
       ay(3000);

       estroy our window. We are passing in the pointer
        points to the memory allocated by the
       CreateWindow' function. Remember, this is
       style' API, we don't have destructors.
       royWindow(window);

       afely uninitialize SDL2, that is, we are
       g down the subsystems here before we exit
       rogram.
       t();

       );
```

- **g++** -- this is the compiler (you can also use clang++ or msvc if you prefer)
- **main.cpp** -- Compilers compile .cpp files (source files).
  - .h and .hpp (header files) are not compiled.
- **-o prog** -- This is the output filename of our executable binary
  - note: windows folks use prog.exe (.exe suffix common on windows)
- **-lSDL3** -- The lowercase ‘L’ tells us we are linking in libSDL3.so
  - ‘lib’ is omitted as part of the name, as well as the suffix on linux.
- Note: Mac and Windows users will likely need an ‘-I’ include path to the header files or you can use **pkg-config** to compile

(Aside) More on compilation and linking

- If you're newer to compiled languages like C or C++, you're going to want to take the time to watch **and try** the exercise to the right.
  - Understanding how to build and link libraries is useful!



In 54 Minutes, Understand the whole C and C++ compilation process
https://www.youtube.com/watch?v=ksJ9bdSX5Yo

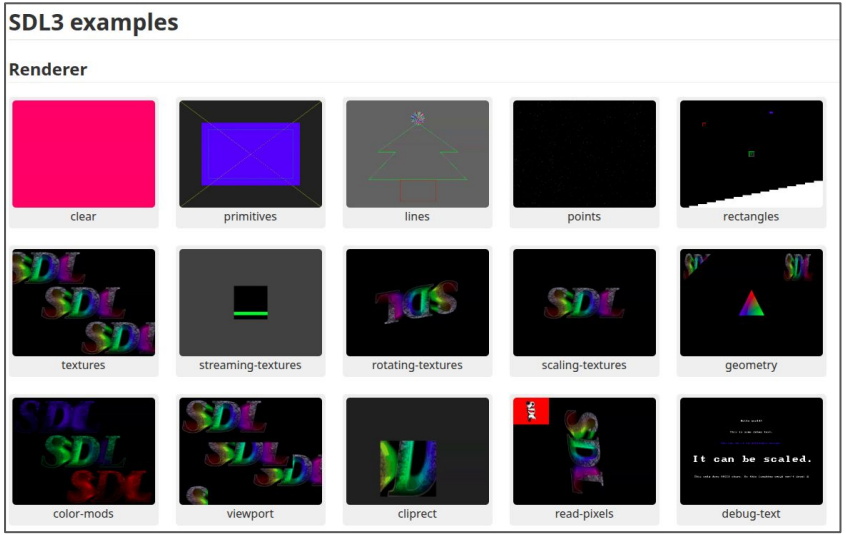
(Aside) Callback based application

- Note:
  - You can also structure your application with ‘callbacks’
  - For platforms (e.g. web, mobile) this may be more appropriate **to not have ‘main()’** as the traditional entry point.
- More information:
  - Docs: https://wiki.libsdl.org/SDL3/SDL_MAIN_USE_CALLBACKS
  - YouTube: SDL3 SDL_AppInit - callback style hello world [SDL3 Episode 6]

```
/*
  Copyright (C) 1997-2025 Sam Lantinga <slouken@libsdl.org>

+--  7 lines: This software is provided 'as-is', without any express or implied-
*/
#define SDL_MAIN_USE_CALLBACKS 1  /* use the callbacks instead of main() */
#include <SDL3/SDL.h>
#include <SDL3/SDL_main.h>

static SDL_Window *window = NULL;
static SDL_Renderer *renderer = NULL;

/* This function runs once at startup. */
SDL_AppResult SDL_AppInit(void **appstate, int argc, char *argv[])
{
+--  6 lines: Create the window ----------------------------------------------
}

/* This function runs when a new event (mouse input, keypresses, etc) occurs. */
SDL_AppResult SDL_AppEvent(void *appstate, SDL_Event *event)
{
+--  5 lines: if (event->type == SDL_EVENT_KEY_DOWN ||-------------------------
}

/* This function runs once per frame, and is the heart of the program. */
SDL_AppResult SDL_AppIterate(void *appstate)
{
+-- 17 lines: const char *message = "Hello World!";-----------------------------

    return SDL_APP_CONTINUE;
}

/* This function runs once at shutdown. */
void SDL_AppQuit(void *appstate, SDL_AppResult result)
{
}
```

(Aside) Emscripten/wasm SDL example

- The SDL 3 examples have all been compiled using emscripten for the web
- I have recorded a tutorial here on how to setup the emscripten toolchain:
  - YouTube: [SDL3 with emscripten for the web [SDL3 Episode 2.8]]
    - Good news -- it was quite easy!
    - Compare below the gcc vs emcc compile commands



https://examples.libsdl.org/SDL3/

```
gcc hello.c -o prog `pkg-config --libs --cflags sdl3`
emsdk$ emcc -sUSE_SDL=3 ../hello.c -o hello.html
```

Basics of Using SDL

Application Structure

Minimal SDL Application

- Here is our minimal SDL Application
- It will pop up a window for 3 seconds on every supported platform

```cpp
// @file: 00.cpp
//
// linux:
// g++ 00.cpp -o prog -lSDL3 && ./prog
// cross-platform:
// g++ 00.cpp -o prog `pkg-config --cflags --libs sdl3` && ./prog
#include <SDL3/SDL.h>

int main(int argc, char *argv[]){
    // Initialization
    SDL_Init(SDL_INIT_VIDEO);

    // Setup one SDL window
    SDL_Window* window;
    window = SDL_CreateWindow("Hello C++ North", 320, 240, SDL_WINDOW_RESIZABLE);

    // Pause program so we can see window for 3 seconds
    SDL_Delay(3000);

    // Destroy any SDL objects we have allocated
    SDL_DestroyWindow(window);

    // Quit SDL and shutdown systems we have initialized
    SDL_Quit();

    return 0;
}
```

(Aside) Error Handling

- Shamefully (so my code fits on slides), I am going to omit most error handling.
- It's worth mentioning 'SDL_GetError()' as the main mechanism for error reporting
  - 'SDL_Log' (and SDL_LogError) otherwise as useful functions
- Most 'creation' functions indicate failure by returning a nullptr.
- Many other functions (e.g. SDL_Init) return a 'bool' value otherwise.
  - For those migrating from SDL2 to SDL3, this is a new and welcomed change in my opinion!

```cpp
// @file: 01.cpp
//
// A few ways to build:
// linux:                   g++ 01.cpp -o prog -lSDL3 && ./prog
// cross-platform:  g++ 01.cpp -o prog `pkg-config --cflags --libs sdl3` && ./prog
#include <SDL3/SDL.h>

int main(int argc, char *argv[]){
    // Initialization
    if (!SDL_Init(SDL_INIT_VIDEO)) {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION,
                     "Couldn't initialize SDL: %s",
                     SDL_GetError());
        return 3;
    }

    // Setup one SDL window
    SDL_Window* window;
    window = SDL_CreateWindow("Hello C++ North", 320, 240, SDL_WINDOW_RESIZABLE);

    if (nullptr == window) {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION,
                     "Couldn't create window and renderer: %s",
                     SDL_GetError());
        return 3;
    }

    // Pause program so we can see window for 3 seconds
    SDL_Delay(3000);

    // Destroy any SDL objects we have allocated
    SDL_DestroyWindow(window);

    // Quit SDL and shutdown systems we have initialized
    SDL_Quit();

    return 0;
}
```

Minimal SDL Application

- So let's advance our "Hello World" SDL application a bit further
- We probably want to build an application that lasts longer than 3 seconds
  - **next slide!**

```cpp
// @file: 00.cpp
//
// linux:
// g++ 00.cpp -o prog -lSDL3 && ./prog
// cross-platform:
// g++ 00.cpp -o prog `pkg-config --cflags --libs sdl3` && ./prog
#include <SDL3/SDL.h>

int main(int argc, char *argv[]){
    // Initialization
    SDL_Init(SDL_INIT_VIDEO);

    // Setup one SDL window
    SDL_Window* window;
    window = SDL_CreateWindow("Hello C++ North", 320, 240, SDL_WINDOW_RESIZABLE);

    // Pause program so we can see window for 3 seconds
    SDL_Delay(3000);

    // Destroy any SDL objects we have allocated
    SDL_DestroyWindow(window);

    // Quit SDL and shutdown systems we have initialized
    SDL_Quit();

    return 0;
}
```

The SDL Event Loop

and

```
union SDL_Event {
        ...
};
```

SDL3 event loop - detecting key presses [SDL3 Episode 7]

Event Loop

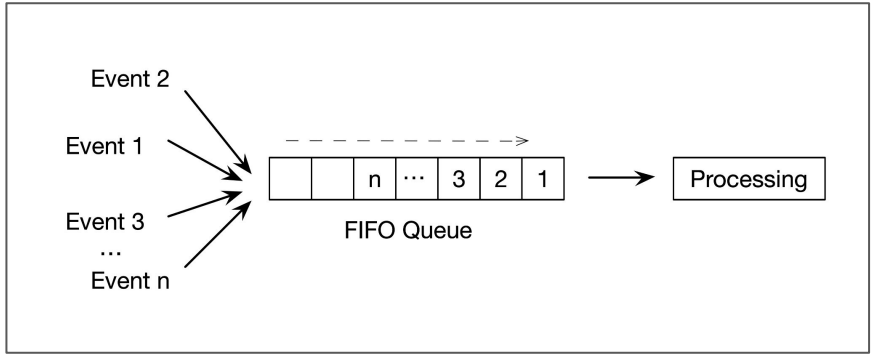
- In this example we have now added an ‘event loop’ into our application
- Now our application will run continuously ‘polling’ for interesting events that happen.
  - Importantly -- our windowed application stays running for more than 3 seconds!

```
1  // @file: 02_event_loop.cpp
2  +--  5 lines: // linux:    ----------------------------
7  #include <SDL3/SDL.h>
8
9  int main(int argc, char *argv[]){
10     // Initialization
11     SDL_Init(SDL_INIT_VIDEO);
12
13     // Setup one SDL window
14     SDL_Window* window;
15     SDL_Event event;
16     window = SDL_CreateWindow("Hello CppCon", 320, 240,
17                               SDL_WINDOW_RESIZABLE);
18
19     // Main application loop
20     while (1) {
21         SDL_PollEvent(&event);
22         if (event.type == SDL_EVENT_QUIT) {
23             break;
24         }
25         if(event.type == SDL_EVENT_KEY_DOWN){
26             SDL_Log("Key was pressed!");
27             // Retrieve the 'virtual scancode'
28             SDL_Log("Keycode: %i",event.key.key);
29         }
30     }
31
32     // Destroy any SDL objects we have allocated
33     SDL_DestroyWindow(window);
34
35     // Quit SDL and shutdown systems we have initialized
36     SDL_Quit();
37
38     return 0;
39 }
```

SDL_PollEvent and SDL_WaitEvent

- Structurally SDL has an ‘internal’ FIFO Queue data structure that processes events
- The next question you might ask -- what exactly are events?
  - i.e.
    - Everything you do (move your mouse, click a button, click the ‘x’ to terminate the application, press a key, etc.) is an ‘event’ processed through this queue.
    - Internally ‘SDL_PollEvent’ calls ‘SDL_WaitEventTimeoutNS’, which will also timeout events after 1ms

```
// Main application loop
while (1) {
    SDL_PollEvent(&event);
    if (event.type == SDL_EVENT_QUIT) {
        break;
    }
    if(event.type == SDL_EVENT_KEY_DOWN){
        SDL_Log("Key was pressed!");
        // Retrieve the 'virtual scancode'
        SDL_Log("Keycode: %i",event.key.key);
    }
}
```



https://i0.wp.com/circuitcellar.com/wp-content/uploads/2020/11/Quick-Bits-0010-Figure-1.png?ssl=1

SDL_Event

- SDL event is a union data type that tracks all of the various SDL events that could take place.
  - You can query the event ‘type’ and then handle it appropriately
    - All ‘Event Types’ start with an unsigned 32-bit int that tells us the type
- https://wiki.libsdl.org/SDL3/SDL_Event

SDL_Event

The structure for all events in SDL.

Header File

Defined in <SDL3/SDL_events.h>

Syntax

```
typedef union SDL_Event
{
    Uint32 type;                    /**< Event type, shared with all events, Uint32 to cover user events which ar
    SDL_CommonEvent common;         /**< Common event data */
    SDL_DisplayEvent display;       /**< Display event data */
    SDL_WindowEvent window;         /**< Window event data */
    SDL_KeyboardDeviceEvent kdevice;/**< Keyboard device change event data */
    SDL_KeyboardEvent key;          /**< Keyboard event data */
    SDL_TextEditingEvent edit;      /**< Text editing event data */
    SDL_TextEditingCandidatesEvent edit_candidates; /**< Text editing candidates event data */
    SDL_TextInputEvent text;        /**< Text input event data */
    SDL_MouseDeviceEvent mdevice;   /**< Mouse device change event data */
    SDL_MouseMotionEvent motion;    /**< Mouse motion event data */
    SDL_MouseButtonEvent button;    /**< Mouse button event data */
    SDL_MouseWheelEvent wheel;      /**< Mouse wheel event data */
    SDL_JoyDeviceEvent jdevice;     /**< Joystick device change event data */
    SDL_JoyAxisEvent jaxis;         /**< Joystick axis event data */
    SDL_JoyBallEvent jball;         /**< Joystick ball event data */
    SDL_JoyHatEvent jhat;           /**< Joystick hat event data */
    SDL_JoyButtonEvent jbutton;     /**< Joystick button event data */
    SDL_JoyBatteryEvent jbattery;   /**< Joystick battery event data */
    SDL_GamepadDeviceEvent gdevice; /**< Gamepad device event data */
    SDL_GamepadAxisEvent gaxis;     /**< Gamepad axis event data */
    SDL_GamepadButtonEvent gbutton; /**< Gamepad button event data */
    SDL_GamepadTouchpadEvent gtouchpad; /**< Gamepad touchpad event data */
    SDL_GamepadSensorEvent gsensor; /**< Gamepad sensor event data */
    SDL_AudioDeviceEvent adevice;   /**< Audio device event data */
    SDL_CameraDeviceEvent cdevice;  /**< Camera device event data */
    SDL_SensorEvent sensor;         /**< Sensor event data */
    SDL_QuitEvent quit;             /**< Quit request event data */
    SDL_UserEvent user;             /**< Custom event data */
    SDL_TouchFingerEvent tfinger;   /**< Touch finger event data */
    SDL_PenProximityEvent pproximity;   /**< Pen proximity event data */
    SDL_PenTouchEvent ptouch;       /**< Pen tip touching event data */
    SDL_PenMotionEvent pmotion;     /**< Pen motion event data */
    SDL_PenButtonEvent pbutton;     /**< Pen button event data */
    SDL_PenAxisEvent paxis;         /**< Pen axis event data */
```

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

SDL_Event - SDL_KeyboardEvent

- As an example, to handle a keyboard event, we see in the ‘SDL_Event’ union that a ‘SDL_EVENT_KEY_DOWN’ event means we have something in ‘key’
  - We can then access the ‘SDL_KeyboardEvent’ fields, to determine the key

```
if(event.type == SDL_EVENT_KEY_DOWN){
    SDL_Log("Key was pressed!");
    // Retrieve the 'virtual scancode'
    SDL_Log("Keycode: %i",event.key.key);
}
```

```
typedef union SDL_Event
{
    Uint32 type;                        /**< Event type, shared with
    SDL_CommonEvent common;             /**< Common event data */
    SDL_DisplayEvent display;           /**< Display event data */
    SDL_WindowEvent window;             /**< Window event data */
    SDL_KeyboardDeviceEvent kdevice;    /**< Keyboard device change
    SDL_KeyboardEvent key;              /**< Keyboard event data */
```

SDL_EVENT_FINGER_MOTION, SDL_EVENT_FINGER_DOWN, SDL_EVENT_FINGER_UP	SDL_TouchFingerEvent	tfinger
SDL_EVENT_KEYBOARD_ADDED, SDL_EVENT_KEYBOARD_REMOVED	SDL_KeyboardDeviceEvent	kdevice
SDL_EVENT_KEY_DOWN, SDL_EVENT_KEY_UP	SDL_KeyboardEvent	key
SDL_EVENT_JOYSTICK_ADDED, SDL_EVENT_JOYSTICK_REMOVED, SDL_EVENT_JOYSTICK_UPDATE_COMPLETE	SDL_JoyDeviceEvent	jdevice

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

SDL_Event - SDL_KeyboardEvent

- As an example, to handle a keyboard event, we see in the 'SDL_Event' union that a 'SDL_EVENT_KEY_D… means we have some… 'key'
  - We can then acc… 'SDL_KeyboardE… to determine th…

```
if(event.type == SDL_EVENT_KEY_DOWN){
    SDL_Log("Key was pressed!");
    // Retrieve the 'virtual scancode'
    SDL_Log("Keycode: %i",event.key.key);
```

```
/**< Event type, shared with
/**< Common event data */
/**< Display event data */
/**< Window event data */
/**< Keyboard device change
/**< Keyboard event data */
```

Great, so we have a **windowed application** that can respond to **events** (e.g. **keypresses**), let's now add graphics!

SDL_EVENT_FINGER_MOTION, SDL_EVENT_FINGER_DOWN, SDL_EVENT_FINGER_UP	SDL_TouchFingerEvent	tfinger
SDL_EVENT_KEYBOARD_ADDED, SDL_EVENT_KEYBOARD_REMOVED	SDL_KeyboardDeviceEvent	kdevice
SDL_EVENT_KEY_DOWN, SDL_EVENT_KEY_UP	SDL_KeyboardEvent	key
SDL_EVENT_JOYSTICK_ADDED, SDL_EVENT_JOYSTICK_REMOVED, SDL_EVENT_JOYSTICK_UPDATE_COMPLETE	SDL_JoyDeviceEvent	jdevice

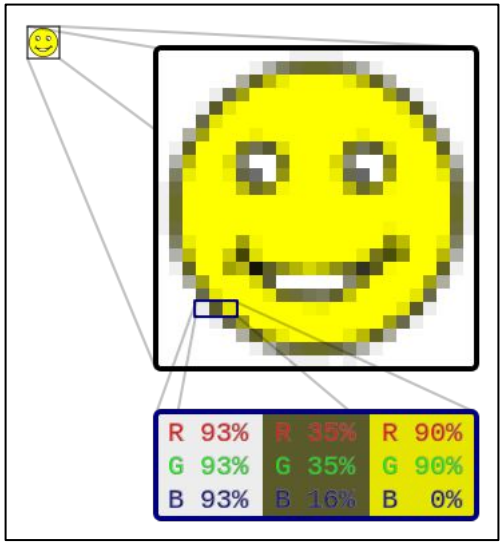
2D Game Terminology

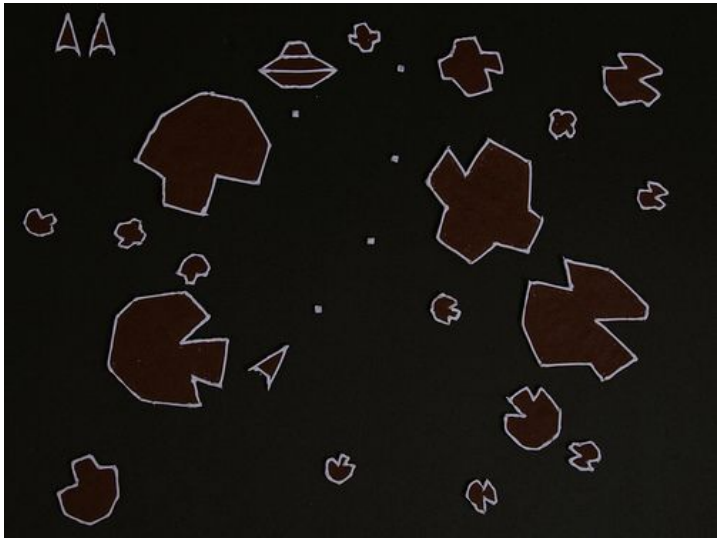
Foundational definitions

2D Rendering

- Rendering is an enormous topic, even when considering just 2D Games
- But let's at least understand the basics so we can use SDL's built-in 2D graphics functions to build a graphics application

In SDL we can use Raster or Vector Graphics

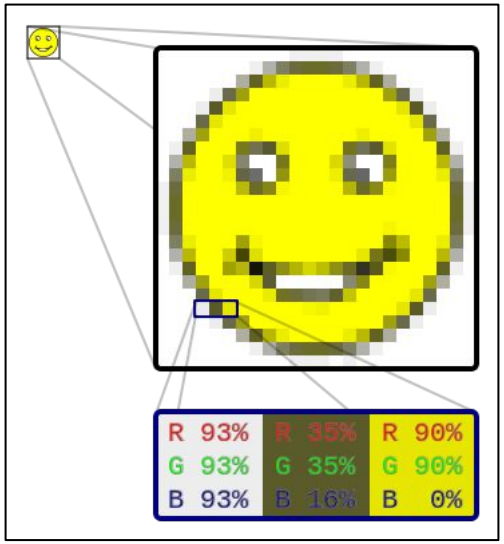


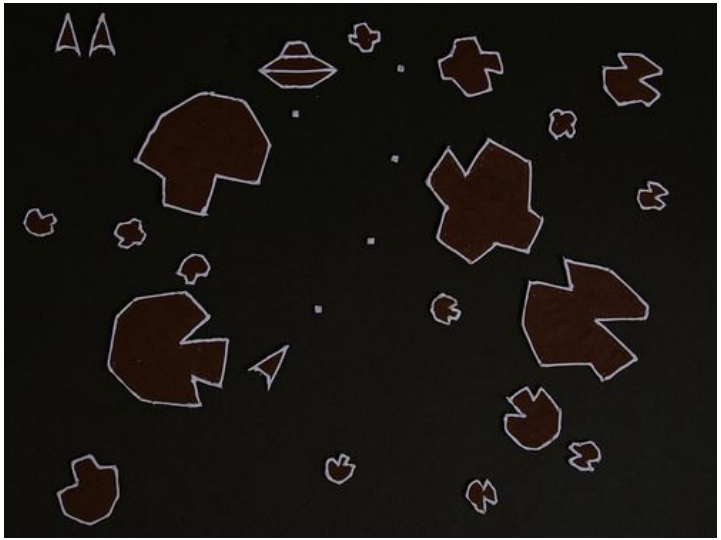


- Raster graphics use individual pixels and are stored in an **SDL_Surface**

- Vector graphics use some mathematical description (e.g. a line function) to draw graphics

In SDL we can use Raster or Vector Graphics





- Pros: Total artist control of every pixel
- Cons: Less scalable, usually much higher memory footprint

- Pros: Scales easily, often very compact representation
- Cons: May be harder to represent any shape

In SDL we can use Raster or Vector Graphics

SDL_BlitSurface

Performs a fast blit from the source surface to the destination surface.

Syntax

```
int SDL_BlitSurface(SDL_Surface *src, const SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstrect);
```

Function Parameters

src	the SDL_Surface structure to be copied from
srcrect	the SDL_Rect structure representing the rectangle to be copied, or NULL to copy the entire surface
dst	the SDL_Surface structure that is the blit target
dstrect	the SDL_Rect structure representing the x and y position in the destination surface. On input the width and height are ignored (taken from srcrect), and on output this is filled in with the actual rectangle used after clipping.

Return Value

Returns 0 on success or a negative error code on failure; call SDL_GetError() for more information.

Remarks

This assumes that the source and destination rectangles are the same size. If either `srcrect` or `dstrect` are NULL, the entire surface (`src` or `dst`) is copied. The final blit rectangles are saved in `srcrect` and `dstrect` after all clipping is performed.

The blit function should not be called on a locked surface.

The blit semantics for surfaces with and without blending and colorkey are defined as follows:

SDL_RenderLine

Draw a line on the current rendering target at subpixel precision.

Syntax

```
int SDL_RenderLine(SDL_Renderer *renderer, float x1, float y1, float x2, float y2);
```

Function Parameters

renderer	The renderer which should draw a line.
x1	The x coordinate of the start point.
y1	The y coordinate of the start point.
x2	The x coordinate of the end point.
y2	The y coordinate of the end point.

Return Value

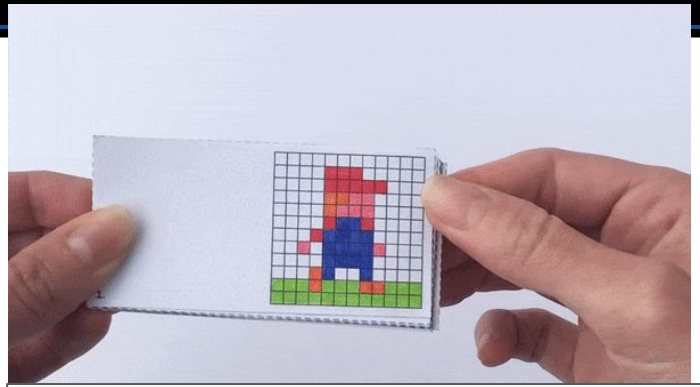
Returns 0 on success, or -1 on error

- We can ‘blit’ or copy on the CPU to update pixels
- Or upload pixels to GPU as textures

- Vector graphics can be batched to draw on the screen

Displaying Raster Graphics to the Screen

- In order to draw raster graphics to the screen, you ‘blit’ (i.e. copy a surface to another surface) every frame in your main game loop.
  - You can imagine this like a ‘flip book’
    - e.g. Every frame you copy some set of pixels to the ‘window surface’
    - If using hardware accelerated graphics (i.e. a GPU) -- then we typically ‘fill’ a buffer using the GPU, and then ‘flip to this buffer when filled.



SDL_BlitSurface

Performs a fast blit from the source surface to the destination surface.

Syntax

```
int SDL_BlitSurface(SDL_Surface *src, const SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstrect);
```

Function Parameters

src	the SDL_Surface structure to be copied from
srcrect	the SDL_Rect structure representing the rectangle to be copied, or NULL to copy the entire surface
dst	the SDL_Surface structure that is the blit target
dstrect	the SDL_Rect structure representing the x and y position in the destination surface. On input the width and height are ignored (taken from srcrect), and on output this is filled in with the actual rectangle used after clipping.

Return Value

Returns 0 on success or a negative error code on failure; call SDL_GetError() for more information.

Remarks

This assumes that the source and destination rectangles are the same size. If either `srcrect` or `dstrect` are NULL, the entire surface (`src` or `dst`) is copied. The final blit rectangles are saved in `srcrect` and `dstrect` after all clipping is performed.

The blit function should not be called on a locked surface.

The blit semantics for surfaces with and without blending and colorkey are defined as follows:

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

Hardware Accelerated 2D Graphics

Using our GPU instead of ‘blitting’ pixels to the screen

2D Game Engine - Hardware Acceleration (1/2)

- Blitting to the screen takes place on the CPU with SDL_BlitSurface
  - When we blit using an SDL_Surface, we are effectively performing a memcpy on the cpu to replace pixels on the window we are drawing to
- Performing lots of CPU memory operations is going to be very costly from a performance standpoint
  - For example, think about if we have several sprites with several frames of animation (see top-right image)

SDL_BlitSurface

Performs a fast blit from the source surface to the destination surface.

Syntax

```
int SDL_BlitSurface(SDL_Surface *src, const SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstrect);
```

Function Parameters

src	the SDL_Surface structure to be copied from
srcrect	the SDL_Rect structure representing the rectangle to be copied, or NULL to copy the entire surface
dst	the SDL_Surface structure that is the blit target
dstrect	the SDL_Rect structure representing the x and y position in the destination surface. On input the width and height are ignored (taken from srcrect), and on output this is filled in with the actual rectangle used after clipping.

Return Value

Returns 0 on success or a negative error code on failure; call SDL_GetError() for more information.

Remarks

This assumes that the source and destination rectangles are the same size. If either `srcrect` or `dstrect` are NULL, the entire surface (`src` or `dst`) is copied. The final blit rectangles are saved in `srcrect` and `dstrect` after all clipping is performed.

The blit function should not be called on a locked surface.

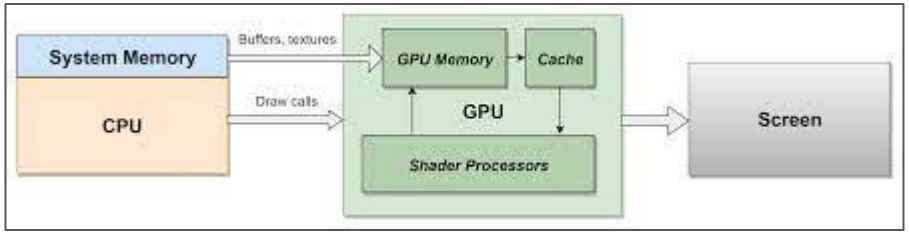
The blit semantics for surfaces with and without blending and colorkey are defined as follows:

2D Game Engine - Hardware Acceleration (2/2)

- Thus to speed up drawing, we want to take advantage of our Graphics Processing Unit (GPU, i.e. Graphics card) to speed up this operation.
  - 2D games use ‘quads’ (two triangles) with a texture to draw the graphics

2D Game Engine - Hardware Acceleration - Textures (1/3)

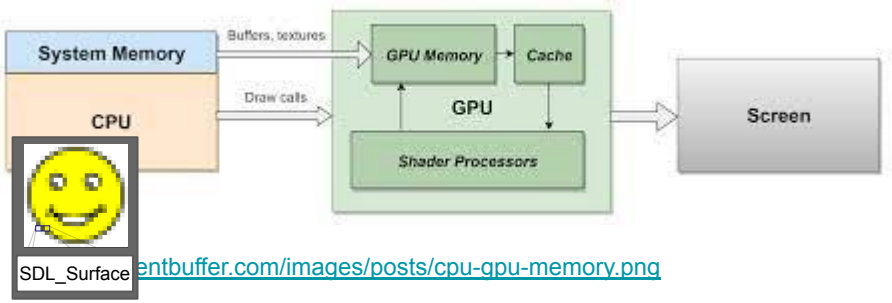
- The first step to taking advantage of our GPU is to create (i.e. allocate memory) on the GPU in a ‘texture’
  - A ‘texture’ is image data that is on the GPU
- Step two is to then allocate the ‘pixels’ onto the GPU in a ‘SDL_Texture’
  - (sometimes you’ll hear pixels on the GPU referred to as ‘texels’)



http://fragmentbuffer.com/images/posts/cpu-gpu-memory.png

2D Game Engine - Hardware Acceleration - Textures (2/3)

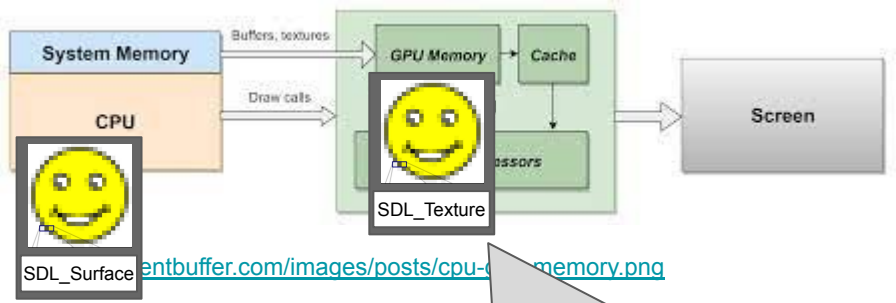
- The first step to taking advantage of our GPU is to create (i.e. allocate memory) on the GPU in a ‘texture’
  - A ‘texture’ is image data that is on the GPU
- Step two is to then allocate the ‘pixels’ onto the GPU in a ‘SDL_Texture’
  - (sometimes you’ll hear pixels on the GPU referred to as ‘texels’)



- Observe an SDL_Surface allocated on the CPU
- (SDL_Surface is a collection of pixels in cpu memory)

2D Game Engine - Hardware Acceleration - Textures (3/3)

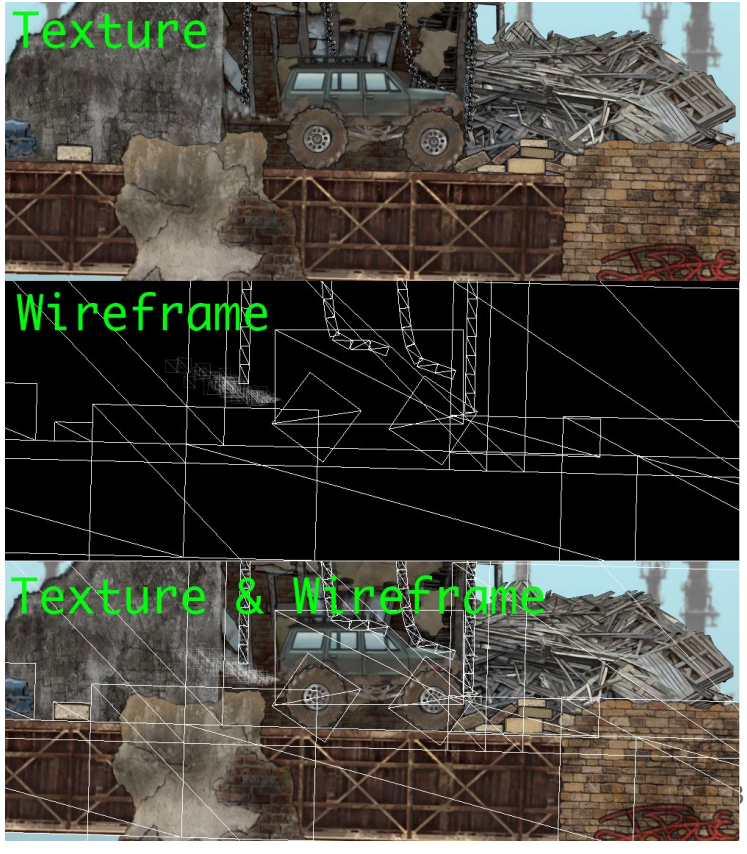
- The first step to taking advantage of our GPU is to create (i.e. allocate memory) on the GPU in a ‘texture’
  - A ‘texture’ is image data that is on the GPU
- Step two is to then allocate the ‘pixels’ onto the GPU in a ‘SDL_Texture’
  - (sometimes you’ll hear pixels on the GPU referred to as ‘texels’)



- When we do SDL_CreateTextureFromSurface, those pixels are now in GPU memory
- Our GPU can display (i.e. copy pixels to video display) much faster from video memory.

2D Engine - Hardware Acceleration - Textures (1/2)

- The ‘textures’ that we draw in 2D games are displayed as ‘quads’ (two triangles) with our texture
  - Observe that some of the pixels are transparent
  - You can view this tutorial on setting transparent pixels using a color key technique
    - SDL3 Color Key on surfaces for transparency [SDL3 Episode 18]



2D Engine - Hardware Acceleration - Textures (2/2)

- Note: Since we are drawing quads, there's nothing stopping us from creating a '3D' game, where we just ignoring the 'z' component of our quad.

SDL Video API

Hardware Accelerated Renderer

SDL3 Hardware Accelerated window - CreateRenderer [SDL3 Episode 13]

Creating a Hardware Renderer

- To draw ‘hardware accelerated graphics’ to the screen with ‘textures’ we need a ‘SDL_Renderer’ object
  - SDL_Renderer is the ‘context’ (state holding object) that will help us draw graphics to our screen using a GPU device
- Note:
  - A fallback software rendering mode (https://wiki.libsdl.org/SDL3/SDL_CreateSoftwareRenderer) can also be used in SDL3 as a fallback

SDL_CreateRenderer

Create a 2D rendering context for a window.

Syntax

```
SDL_Renderer * SDL_CreateRenderer(SDL_Window * window,
                                  int index, Uint32 flags);
```

Function Parameters

window	the window where rendering is displayed
index	the index of the rendering driver to initialize, or -1 to initialize the first one supportir
flags	0, or one or more SDL_RendererFlags OR'd together

Return Value

Returns a valid rendering context or NULL if there was an error; call SDL_GetError() for more information.

Hardware Renderer (1/2)

- The following example demonstrates creating a hardware accelerated renderer
- We can now draw graphics using a a GPU accelerated graphics API
- Note:
  - Note: OpenGL was the default selected for my machine
  - You can explicitly set the renderer (e.g. to ‘vulkan’) when creating the renderer or query available renders [SDL3 Hardware Accelerated window - CreateRenderer [SDL3 Episode 13]]

```
1  // @file: 03_renderer.cpp
2  +--  5 lines: // linux:    ----------
7  #include <SDL3/SDL.h>
8
9  int main(int argc, char *argv[]){
10     // structures
11     SDL_Window *window;
12     SDL_Renderer *renderer;
13     SDL_Event event;
14
15     // Initialization
16     SDL_Init(SDL_INIT_VIDEO);
17     SDL_CreateWindowAndRenderer("Hello CppCon", 320, 240,
18                                 SDL_WINDOW_RESIZABLE, &window, &renderer)
19     SDL_Log("Renderer: %s",SDL_GetRendererName(renderer));
20
21     // Main application loop
22     int state=0;
23     while (1) {
24         SDL_PollEvent(&event);
25         if (event.type == SDL_EVENT_QUIT) {
26             break;
27         }
28
29         SDL_RenderClear(renderer);
30         SDL_SetRenderDrawColor(renderer, 0xFF,0x00,0x0,0xFF);
31         SDL_RenderPresent(renderer);
32     }
33
34     // Explicit cleanup of allocated resources
35     SDL_DestroyRenderer(renderer);
36     SDL_DestroyWindow(window);
37     SDL_Quit();
38
39     return 0;
40 }
```

Hardware Renderer (2/2)

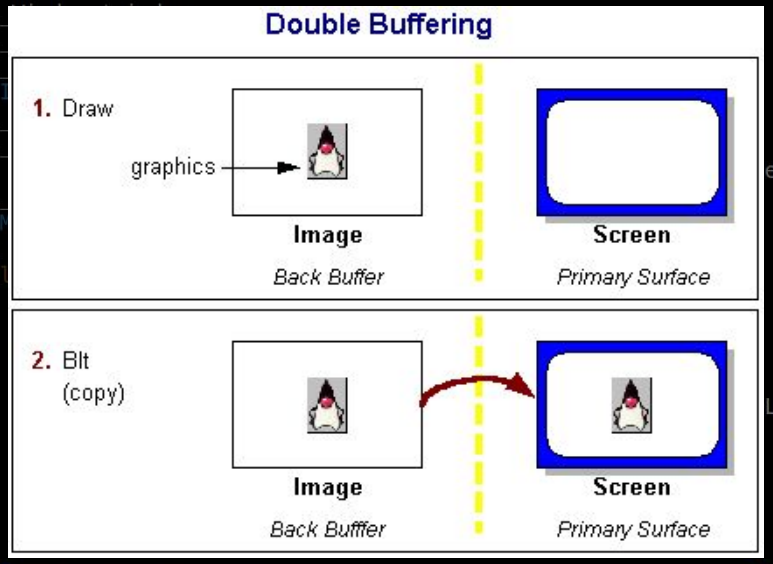
- Combining what we previously learned, we can add some ‘input’ and change the background color with the ‘R’ key.
- I’ve highlighted a few of the SDL_Render functions to test that our renderer has been created

```
1  // @file: 04_events_renderer.cpp
2  +--  5 lines: // linux:    ----------------------------------------------
7  #include <SDL3/SDL.h>
8  int main(int argc, char *argv[]){
9      // structures
10     SDL_Window *window;
11     SDL_Renderer *renderer;
12     SDL_Event event;
13     // Initialization
14     SDL_Init(SDL_INIT_VIDEO);
15     SDL_CreateWindowAndRenderer("Hello CppCon", 320, 240,
16                                 SDL_WINDOW_RESIZABLE, &window, &renderer);
17     SDL_Log("Renderer: %s",SDL_GetRendererName(renderer));
18     // Main application loop
19     int state=0;
20     while (1) {
21         if(SDL_PollEvent(&event)){
22             if (event.type == SDL_EVENT_QUIT) {
23                 break;
24             }
25             if(event.type == SDL_EVENT_KEY_DOWN && event.key.key==SDLK_R){
26                 state =1;
27             }else if(event.type == SDL_EVENT_KEY_DOWN){
28                 state =0;
29             }
30         }
31
32         SDL_RenderClear(renderer);
33
34         if(state){
35             SDL_SetRenderDrawColor(renderer, 0xFF,0x00,0x0,0xFF);
36         }else{
37             SDL_SetRenderDrawColor(renderer, 0xFF,0xFF,0x0,0xFF);
38         }
39
40         SDL_RenderPresent(renderer);
41     }
42     // Explicit cleanup of allocated resources
43     SDL_DestroyRenderer(renderer);
44     SDL_DestroyWindow(window);
45     SDL_Quit();
```

(Aside) Double Buffering (1/3)

- SDL handles for us something known as '**double buffering**'
  - Basically this is a way to prevent a flickering effect if we directly to a buffer (i.e. single buffering)
  - Thus to fix this flickering one common strategy is to use a 'second buffer'
    - i.e. You draw to a 'backbuffer' and then when the contents are filled, 'flip' (or swap) the buffer to the front.

Note: An array is often referred to as a 'buffer' (a buffer is just something that stores bytes of data)

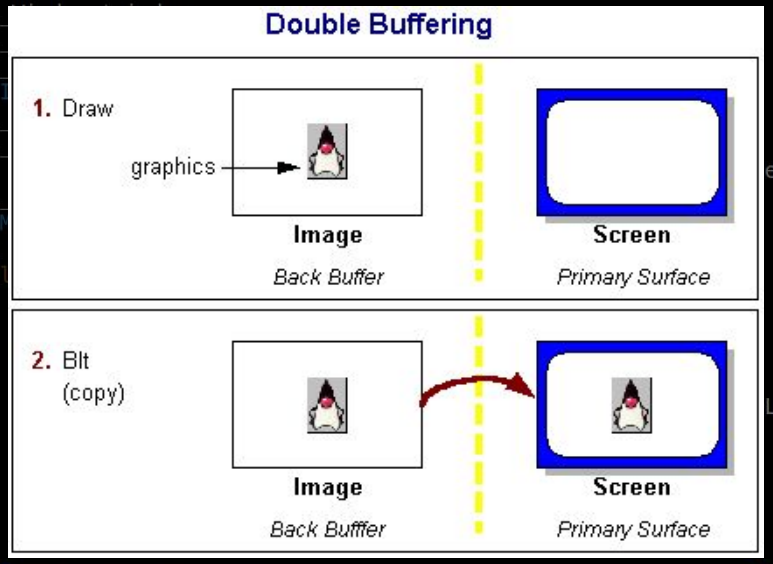



```
// @file: 04.cpp
+--  5 lines: // linux:
#include <SDL3/SDL.h>
int main(int argc, char *argv[]){
    // structures
    SDL_
    SDL_
    SDL_
    //
    SDL_
    SDL_
    SDL_
    //
    int
    whi
        SDL_RenderClear(renderer);
        if(state){
            SDL_SetRenderDrawColor(renderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(renderer, 0xFF,0xFF,0x0,0xFF);
        }
        SDL_RenderPresent(renderer);
    }
    // Explicit cleanup of allocated resources
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
```

(Aside) Double Buffering (2/3)

- More on the double buffering strategy:
  - https://gameprogrammingpatterns.com/double-buffer.html
- Note: Separating computation into two steps can be useful for non-graphics techniques as well.
- Note: Some 3D graphics techniques also use ‘triple buffering’
  - https://www.anandtech.com/show/2794/2

Note: An array is often referred to as a ‘buffer’ (a buffer is just something that stores bytes of data)



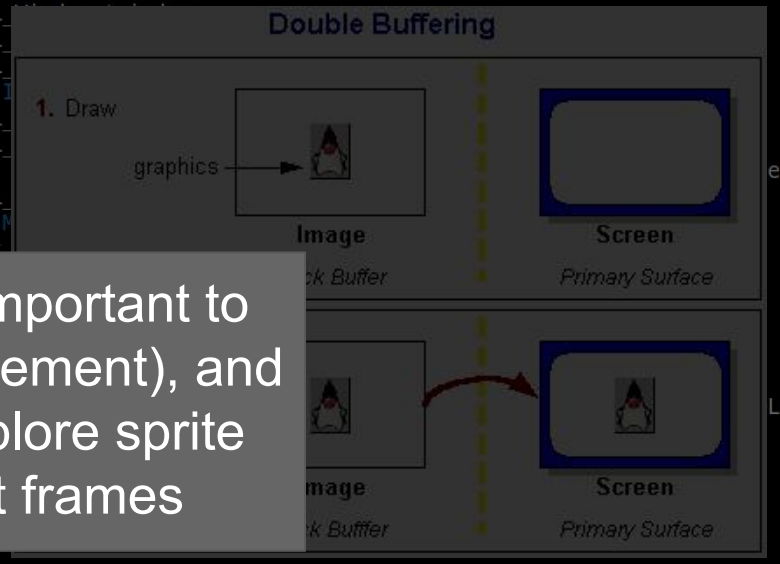

```
// @file: 04.cpp
+--  5 lines: // linux:
#include <SDL3/SDL.h>
int main(int argc, char *argv[]){
    // structures
    SDL_
    SDL_
    SDL_
    //
    SDL_
    SDL_
    SDL_
    //
    int
    whi
        SDL_RenderClear(renderer);
        if(state){
            SDL_SetRenderDrawColor(renderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(renderer, 0xFF,0xFF,0x0,0xFF);
        }
        SDL_RenderPresent(renderer);
    }
    // Explicit cleanup of allocated resources
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
```

(Aside) Double Buffering (3/3)

- More on the double buffering strategy:
  - https://gameprogrammingpatterns.com
- Note: Sep… computat… can be us… non-grap… well.
- Note: Some 3D graphics techniques also use ‘triple buffering’
  - https://www.anandtech.com/show/2794/2

Note: An array is often referred to as a ‘buffer’ (a buffer is just something that stores bytes of data)

Double buffering is very important to animation (any type of movement), and in the future we’ll also explore sprite animation with different frames




```
// @file: 04.cpp
+--  5 lines: // linux:
#include <SDL3/SDL.h>
int main(int argc, char *argv[]){
    // structures
    SDL
    SDL
    SDL
    // I
    SDL
    SDL
    SDL
    // M
    int
                                                    erer);
                                                    LK_R){
        SDL_RenderClear(renderer);

        if(state){
            SDL_SetRenderDrawColor(renderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(renderer, 0xFF,0xFF,0x0,0xFF);
        }

        SDL_RenderPresent(renderer);
    }
    // Explicit cleanup of allocated resources
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
```

SDL Application Structure

Some light structuring (and resource management)

struct SDLApplication

- Quite soon, our application is going to start getting quite large
- I generally recommend using a ‘struct’ of some sort to start encapsulating ‘state’ and the ‘mainloop’ of your game/application
  - This is testable
    - i.e. we can add parameters to ‘app’ to change behavior or otherwise record ‘state’ in one location
  - It keeps our main function clean
  - We can do interesting things like add ‘callbacks’ and start to control a bit more how data is used and loaded.

```
1  // @file: 05.cpp
2  +--  5 lines: // linux:    ----------
7  #include <SDL3/SDL.h>
8
9  struct SDLApplication{
10     SDL_Window *window;
11     SDL_Renderer *renderer;
12 +--  6 lines: SDLApplication(){-----
18 +--  3 lines: void Initialization(){
21
22 +--  6 lines: void Cleanup(){-------
28
29 +-- 26 lines: void MainLoop(){------
55 };
56
57 int main(int argc, char *argv[]){
58     SDLApplication app;
59     app.Initialization();
60     app.MainLoop();
61     app.Cleanup();
62
63     return 0;
64 }
```

SDL Video API

A brief recap on terms and displaying a texture

SDL Video API Basics

At the very least there are three things in the Video API of SDL we need to know about

1. Window
   - Which we have covered the basics of
2. Surfaces
   - Mechanism for storing pixels
3. Textures
   - pixel data stored on the GPU

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

SDL 3.0 API by Category

Basics

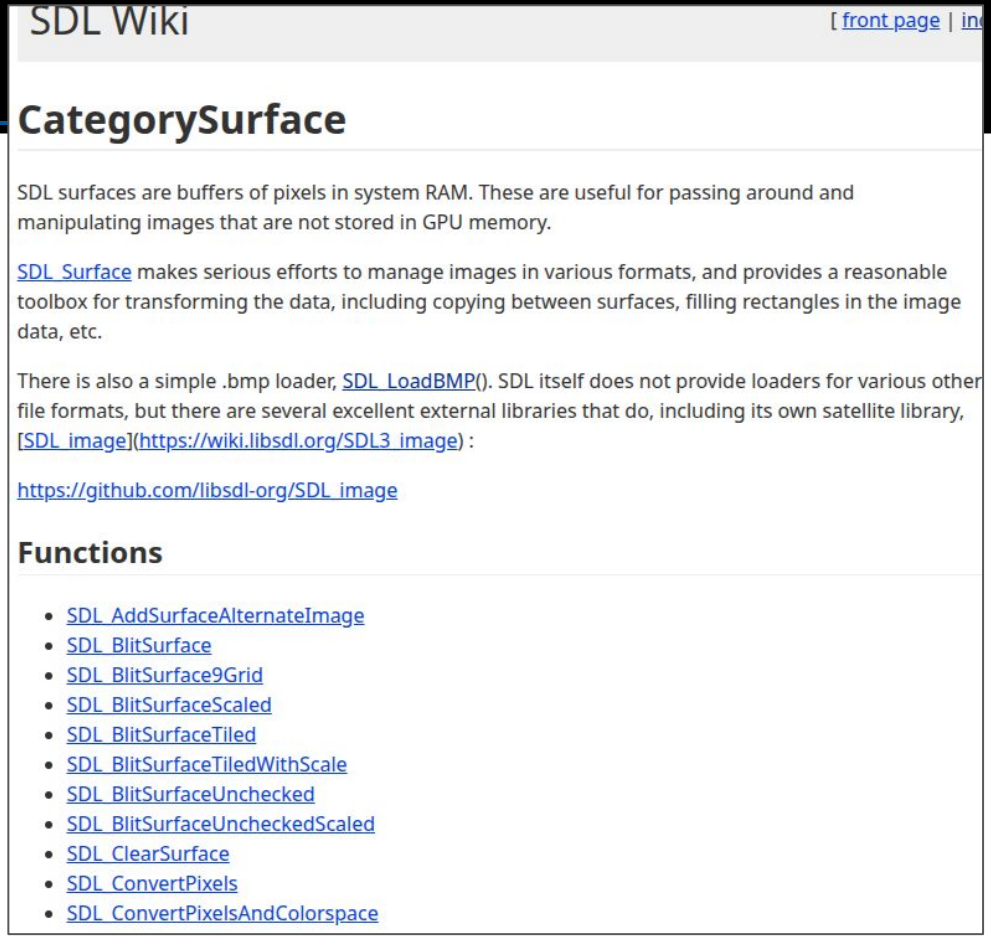
View information and functions related to...	View the header
Application entry points	SDL_main.h
Initialization and Shutdown	SDL_init.h
Configuration Variables	SDL_hints.h
Object Properties	SDL_properties.h
Error Handling	SDL_error.h
Log Handling	SDL_log.h
Assertions	SDL_assert.h
Querying SDL Version	SDL_version.h

Video

View information and functions related to...	View the header
Display and Window Management	SDL_video.h
2D Accelerated Rendering	SDL_render.h
Pixel Formats and Conversion Routines	SDL_pixels.h
Blend modes	SDL_blendmode.h
Rectangle Functions	SDL_rect.h
Surface Creation and Simple Drawing	SDL_surface.h
Clipboard Handling	SDL_clipboard.h
Vulkan Support	SDL_vulkan.h
Metal Support	SDL_metal.h

SDL_Surface (1/2)

- Surfaces in SDL allow us to hold pixels.
  - Within a surface we additionally have the ability to manipulate or copy pixels as needed.
- By default, SDL supports the loading of .bmp (bitmap) based images as well
  - This will be enough to get us started!



SDL Wiki [front page | in

CategorySurface

SDL surfaces are buffers of pixels in system RAM. These are useful for passing around and manipulating images that are not stored in GPU memory.

SDL_Surface makes serious efforts to manage images in various formats, and provides a reasonable toolbox for transforming the data, including copying between surfaces, filling rectangles in the image data, etc.

There is also a simple .bmp loader, SDL_LoadBMP(). SDL itself does not provide loaders for various other file formats, but there are several excellent external libraries that do, including its own satellite library, [SDL_image](https://wiki.libsdl.org/SDL3_image) :

https://github.com/libsdl-org/SDL_image

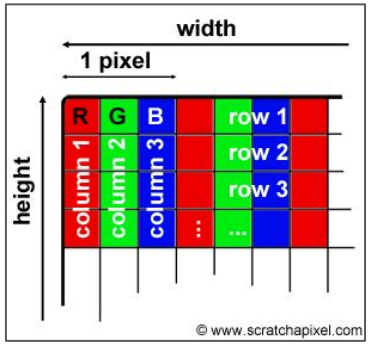
Functions

- SDL_AddSurfaceAlternateImage
- SDL_BlitSurface
- SDL_BlitSurface9Grid
- SDL_BlitSurfaceScaled
- SDL_BlitSurfaceTiled
- SDL_BlitSurfaceTiledWithScale
- SDL_BlitSurfaceUnchecked
- SDL_BlitSurfaceUncheckedScaled
- SDL_ClearSurface
- SDL_ConvertPixels
- SDL_ConvertPixelsAndColorspace

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

SDL_Surface (2/2)

- SDL_Surface itself is a struct, the important part for us is the format of the image data and the width and height of the image
  - As a quick refresher, images are made up of individual pixels.
  - Each pixel has a red, green, and blue color channel
    - (Observe on the figure in the bottom-right)
- Note: Some color formats also contain an ‘alpha’ (A) component for the transparency/opacity of the pixel.



SDL_Surface

A collection of pixels used in software blitting.

Header File

Defined in <SDL3/SDL_surface.h>

Syntax

```
struct SDL_Surface
{
    SDL_SurfaceFlags flags;     /*
    SDL_PixelFormat format;     /*
    int w;                      /*
    int h;                      /*
    int pitch;                  /*
    void *pixels;               /*

    int refcount;               /*

    void *reserved;             /*
};
```

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

Hardware Accelerated Rendering

- Blitting is a useful technique, but copying all the pixels per SDL_Surface to the screen is slow
- Instead we have seen that it's easy to setup a SDL_Renderer which is GPU accelerated
  - (SDL_Renderer again is built on top of OpenGL, Vulkan, D3D, Metal, etc.)
- **A texture** is a surface (i.e pixels) that has been transferred to the GPU
  - Video memory is much faster to display -- because the graphics card can do the hard work.
- https://wiki.libsdl.org/SDL3/CategoryRender

2D Accelerated Rendering

Include File(s): SDL_render.h

Introduction

This category contains functions for 2D accelerated rendering.

This API supports the following features:

single pixel points

single pixel lines

filled rectangles

texture images

All of these may be drawn in opaque, blended, or additive modes.

The texture images can have an additional color tint or alpha modulation applied to them, and may also be stretched with linear interpolation, rotated or flipped/mirrored.

For advanced functionality like particle effects or actual 3D you should use SDL's OpenGL/Direct3D support or one of the many available 3D engines.

This API is not designed to be used from multiple threads, see SDL issue #986 for details.

SDL_BlendFactor

SDL_BlendOperation

SDL_Renderer

SDL_RendererFlags

SDL_RendererFlip

SDL_RendererInfo

SDL_Texture

SDL_TextureAccess

SDL_TextureModulate

SDL_Texture

- Our goal now will be to load on the cpu a surface, and then at run-time (i.e. while our program is running), upload data to the GPU and create a texture.
- Games use ‘textures’ (video memory) to render quickly by using the GPU (Device) that is specifically designed to draw pixels from video memory fast

SDL_Texture

A structure that contains an efficient, driver-specific representation of pixel data.

Related Functions

SDL_CreateTexture
SDL_CreateTextureFromSurface
SDL_DestroyTexture
SDL_GetTextureAlphaMod
SDL_GetTextureBlendMode
SDL_GetTextureColorMod
SDL_LockTexture
SDL_QueryTexture
SDL_RenderCopy
SDL_SetTextureAlphaMod
SDL_SetTextureBlendMode
SDL_SetTextureColorMod
SDL_UnlockTexture
SDL_UpdateTexture

Using the Render API

- In order to make use of hardware acceleration, we are going to use the SDL_Renderer
  - An SDL_Renderer contains the rendering state and the rendering target
    - This is what effectively manages all of our GPU based textures (or other primitives we might draw) in video memory.
  - This is just using a layer of abstraction over D3D, OpenGL, etc.
    - https://github.com/libsdl-org/SDL/blob/main/src/render/SDL_render.c

SDL_Renderer

A structure that contains a rendering state.

Related Functions

SDL_CreateRenderer
SDL_CreateSoftwareRenderer
SDL_CreateTexture
SDL_CreateTextureFromSurface
SDL_CreateWindowAndRenderer
SDL_DestroyRenderer
SDL_GetRenderDrawBlendMode
SDL_GetRenderDrawColor
SDL_GetRendererInfo
SDL_GetRendererOutputSize
SDL_GetRenderTarget
SDL_RenderClear
SDL_RenderCopy
SDL_RenderCopyEx
SDL_RenderDrawLine
SDL_RenderDrawLines
SDL_RenderDrawPoint
SDL_RenderDrawPoints
SDL_RenderDrawRect

3 Functions of Interest to Start

- For our Render loop, we'll look at 3 main functions to get started
- SDL_RenderClear(. . .)
  - Clear our renderer to an empty screen
- SDL_RenderTexture(. . .)
  - Copy a texture into our render target (i.e. where we want to draw
- SDL_RenderPresent(. . .)
  - Display everything that has been copied to our renderer since we last cleared it.
  - Again -- this takes everything in the 'backbuffer' (e.g. SDL_RenderTexture) and presents it to the 'frontbuffer'

SDL_Renderer

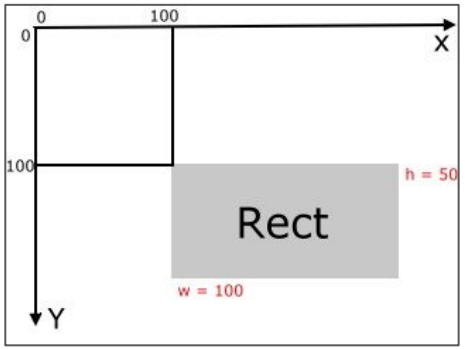
A structure that contains a rendering state.

Related Functions

SDL_CreateRenderer
SDL_CreateSoftwareRenderer
SDL_CreateTexture
SDL_CreateTextureFromSurface
SDL_CreateWindowAndRenderer
SDL_DestroyRenderer
SDL_GetRenderDrawBlendMode
SDL_GetRenderDrawColor
SDL_GetRendererInfo
SDL_GetRendererOutputSize
SDL_GetRenderTarget
SDL_RenderClear
SDL_RenderCopy
SDL_RenderCopyEx
SDL_RenderDrawLine
SDL_RenderDrawLines
SDL_RenderDrawPoint
SDL_RenderDrawPoints
SDL_RenderDrawRect

SDL_Rect and SDL_FRect

- Note, one other struct built into SDL is SDL_Rect
  - (There is also SDL_FRect for floats)
- This will often be useful for for positioning elements, or selecting regions of interest.



https://upload.wikimedia.org/wikibooks/de/archive/1/1e/20060304102840%21SDL_Rect.jpg

SDL_Rect

A structure that contains the definition of a rectangle, with the origin at the upper left.

Data Fields

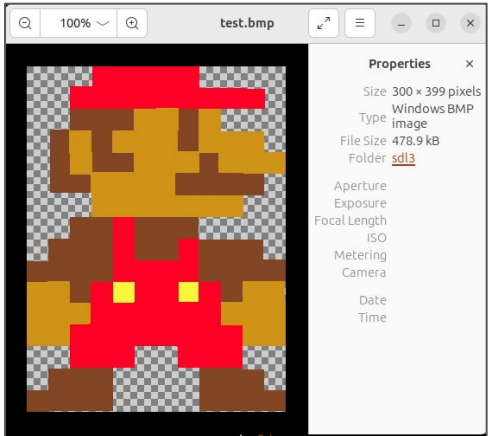
int	**x**	the x location of the rectangle's upper left corner
int	**y**	the y location of the rectangle's upper left corner
int	**w**	the width of the rectangle
int	**h**	the height of the rectangle

Code Examples

```
SDL_Rect srcrect;
SDL_Rect dstrect;

srcrect.x = 0;
srcrect.y = 0;
srcrect.w = 32;
srcrect.h = 32;
dstrect.x = 640/2;
dstrect.y = 480/2;
dstrect.w = 32;
dstrect.h = 32;

SDL_BlitSurface(src, &srcrect, dst, &dstrect);
```

Texture Setup (1/2)

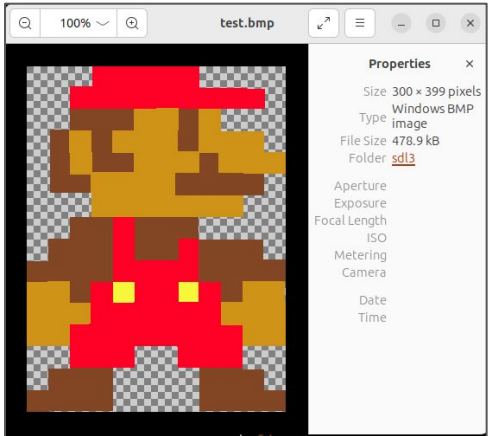
- The following shows how to create a texture on the GPU from a bitmap image.




```
1  // @file: 06_texture.cpp
2  +--  5 lines: // linux:    ----------------------------------------
7  #include <SDL3/SDL.h>
8
9  struct SDLApplication{
10     SDL_Window *mWindow;
11     SDL_Renderer *mRenderer;
12     SDL_Surface* mSurface;
13     SDL_Texture* mTexture;
14
15     SDLApplication(){
16         // Initialization
17         SDL_CreateWindowAndRenderer("Hello CppCon", 320, 240,
18                 SDL_WINDOW_RESIZABLE, &mWindow, &mRenderer);
19         SDL_Log("Renderer: %s",SDL_GetRendererName(mRenderer));
20     }
21     void Initialization(){
22         SDL_Init(SDL_INIT_VIDEO);
23
24         mSurface = SDL_LoadBMP("test.bmp");
25         mTexture = SDL_CreateTextureFromSurface(mRenderer,mSurface);
26         SDL_DestroySurface(mSurface);
27     }
28
29     void Cleanup(){
30         // Explicit cleanup of allocated resources
31         SDL_DestroyTexture(mTexture);
32         SDL_DestroyRenderer(mRenderer);
33         SDL_DestroyWindow(mWindow);
34         SDL_Quit();
35     }
```

Texture Setup (2/2)

- The following shows how to create a texture on the GPU from a bitmap image.




```
1  // @file: 06_texture.cpp
2  +--  5 lines: // linux:
7  #include <SDL3/SDL.h>
8
9  struct SDLApplication{
10     SDL_Window *mWindow;
11     SDL_Renderer *mRende
12     SDL_Surface* mSurfac
13     SDL_Texture* mTextur
14
15     SDLApplication(){
16         // Initialization
17         SDL_CreateWindowAndRenderer("Hello CppCon", 32
18                 SDL_WINDOW_RESIZABLE, &mWindow, &mRender
19         SDL_Log("Renderer: %s",SDL_GetRendererName(mRender
20     }
21     void Initialization(){
22         SDL_Init(SDL_INIT_VIDEO);
23
24         mSurface = SDL_LoadBMP("test.bmp");
25         mTexture = SDL_CreateTextureFromSurface(mRenderer,mSurface);
26         SDL_DestroySurface(mSurface);
27     }
28
29     void Cleanup(){
30         // Explicit cleanup of allocated resources
31         SDL_DestroyTexture(mTexture);
32         SDL_DestroyRenderer(mRenderer);
33         SDL_DestroyWindow(mWindow);
34         SDL_Quit();
35     }
```

Note: I call `SDL_DestroySurface(mSurface)` immediately after I create the texture.

I don't need the pixels (on the CPU) anymore once they are uploaded to the GPU and in an 'SDL_Texture'

Texture

- Putting this all together we have a textured object
- We can then start to form some abstraction around ‘moving’ different instances of this object to make a game.
  - i.e. the ‘rect’ .x and .y values are what will position the texture in our window.

```
void MainLoop(){
    // Main application loop
    int state=0;

    SDL_FRect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = 64;
    rect.h = 64;

    while (1) {
        SDL_Event event;
10 lines: while(SDL_PollEvent(&event)){----------------------------

        SDL_RenderClear(mRenderer);

        if(state){
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0xFF,0x0,0xFF);
        }

        // Render a texture, seleting the texels based on
        // the rectangle.
        SDL_RenderTexture(mRenderer, mTexture, nullptr, &rect);

        // Present all 'drawing' operations that have
        // been queued up to this point.
        SDL_RenderPresent(mRenderer);
    }
}
```

Animated Texture

- SDL_FRect src_rect;
- rect.x = 0;
- rect.y = 0;
- rect.w = 64;
- rect.h = 64;

```
void MainLoop(){
    // Main application loop
    int state=0;

    SDL_FRect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = 64;
    rect.h = 64;

    while (1) {
        SDL_Event event;
10 lines: while(SDL_PollEvent(&event)){----------------------------

        SDL_RenderClear(mRenderer);

        if(state){
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0xFF,0x0,0xFF);
        }

        // Render a texture, seleting the texels based on
        // the rectangle.
        SDL_RenderTexture(mRenderer, mTexture, &src_rect  &rect);

        // Present all 'drawing' operations that have
        // been queued up to this point.
        SDL_RenderPresent(mRenderer);
    }
}
```

Animated Texture

- SDL_FRect src_rect;
- rect.x = 64;
- rect.y = 0;
- rect.w = 64;
- rect.h = 64;

```
void MainLoop(){
    // Main application loop
    int state=0;

    SDL_FRect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = 64;
    rect.h = 64;

    while (1) {
        SDL_Event event;
10 lines: while(SDL_PollEvent(&event)){----------------------------

        SDL_RenderClear(mRenderer);

        if(state){
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0xFF,0x0,0xFF);
        }

        // Render a texture, seleting the texels based on
        // the rectangle.
        SDL_RenderTexture(mRenderer, mTexture, &src_rect  &rect);

        // Present all 'drawing' operations that have
        // been queued up to this point.
        SDL_RenderPresent(mRenderer);
    }
}
```

Animated Texture

- SDL_FRect src_rect;
- rect.x = 128;
- rect.y = 0;
- rect.w = 64;
- rect.h = 64;

```
void MainLoop(){
    // Main application loop
    int state=0;

    SDL_FRect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = 64;
    rect.h = 64;

    while (1) {
        SDL_Event event;
10 lines: while(SDL_PollEvent(&event)){----------------------------

        SDL_RenderClear(mRenderer);

        if(state){
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0xFF,0x0,0xFF);
        }

        // Render a texture, seleting the texels based on
        // the rectangle.
        SDL_RenderTexture(mRenderer, mTexture, &src_rect  &rect);

        // Present all 'drawing' operations that have
        // been queued up to this point.
        SDL_RenderPresent(mRenderer);
    }
}
```

Animated Texture

- SDL_FRect src_rect;
- rect.x = 192;
- rect.y = 0;
- rect.w = 64;
- rect.h = 64;

```
void MainLoop(){
    // Main application loop
    int state=0;

    SDL_FRect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = 64;
    rect.h = 64;

    while (1) {
        SDL_Event event;
10 lines: while(SDL_PollEvent(&event)){----------------------------

        SDL_RenderClear(mRenderer);

        if(state){
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0xFF,0x0,0xFF);
        }

        // Render a texture, seleting the texels based on
        // the rectangle.
        SDL_RenderTexture(mRenderer, mTexture, &src_rect  &rect);

        // Present all 'drawing' operations that have
        // been queued up to this point.
        SDL_RenderPresent(mRenderer);
    }
}
```

Animated Texture

- SDL_FRect src_rect;
- rect.x = 256;
- rect.y = 0;
- rect.w = 64;
- rect.h = 64;

```
void MainLoop(){
    // Main application loop
    int state=0;

    SDL_FRect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = 64;
    rect.h = 64;

    while (1) {
        SDL_Event event;
10 lines: while(SDL_PollEvent(&event)){----------------------------

        SDL_RenderClear(mRenderer);

        if(state){
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0xFF,0x0,0xFF);
        }

        // Render a texture, seleting the texels based on
        // the rectangle.
        SDL_RenderTexture(mRenderer, mTexture, &src_rect  &rect);

        // Present all 'drawing' operations that have
        // been queued up to this point.
        SDL_RenderPresent(mRenderer);
    }
}
```

Animated Texture

- SDL_FRect src_rect;
- rect.x = 320;
- rect.y = 0;
- rect.w = 64;
- rect.h = 64;

```
void MainLoop(){
    // Main application loop
    int state=0;

    SDL_FRect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = 64;
    rect.h = 64;

    while (1) {
        SDL_Event event;
10 lines: while(SDL_PollEvent(&event)){----------------------------

        SDL_RenderClear(mRenderer);

        if(state){
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0x00,0x0,0xFF);
        }else{
            SDL_SetRenderDrawColor(mRenderer, 0xFF,0xFF,0x0,0xFF);
        }

        // Render a texture, seleting the texels based on
        // the rectangle.
        SDL_RenderTexture(mRenderer, mTexture, &src_rect  &rect);

        // Present all 'drawing' operations that have
        // been queued up to this point.
        SDL_RenderPresent(mRenderer);
    }
}
```

A nicer texture animation

- Here's an animation of a sprite
  - Again, selecting a portion of an image from a 'sprite sheet'
  - This is one SDL_Texture
    - (For various reasons, we try to pack sprites into one texture to avoid swapping in new textures on the GPU)
- Note:
  - I use a 'color key' to treat the pink pixels as transparent in this example.
- Note:
  - I am also doing some 'window resizing' and pixel sampling in this example.

SDL3 Texture Sampling and SDL3 upgrade [SDL3 Episode 21]

Plenty more to look at!

- Blending modes
- Color Keying / Transparency
- Pixel and texture manipulation
- Texture Scrolling
- Texture Rotations
- Other APIs of SDL
  - Sound
  - Input
  - Camera Library
  - Networking
  - Image libraries
  - Font libraries
- At this point however -- you have enough to write a game

Abstraction Caution

- It is possible to introduce some abstraction and utilize RAII
  - This way we can ensure to always ‘free’ a texture for instance.
- **However:** We do have to be a little careful with the lifetime of resources however
  - RAII generally can work well, but keep in mind we have resources that live on different devices (i.e. CPU and GPU)
- **I recommend a ‘manager’ class** to otherwise manage the lifetimes of resources like textures.
  - Otherwise, std::shared_ptr<SDL_Texture> is an easy solution.

```
// @file: abstraction.cpp
//
// linux:
// g++ abstraction.cpp -o prog -lSDL3 && ./prog
// cross-platform:
// g++ abstraction.cpp -o prog `pkg-config --cflags --libs sdl3` && ./prog
#include <SDL3/SDL.h>

struct Texture{
    SDL_Texture* mTexture;
    Texture(SDL_Renderer* r, const char* filename){
        SDL_Surface* surface = SDL_LoadBMP(filename);
        mTexture = SDL_CreateTextureFromSurface(r,surface);
        SDL_DestroySurface(surface);
    }
    ~Texture(){
        SDL_DestroyTexture(mTexture);
    }
};
```

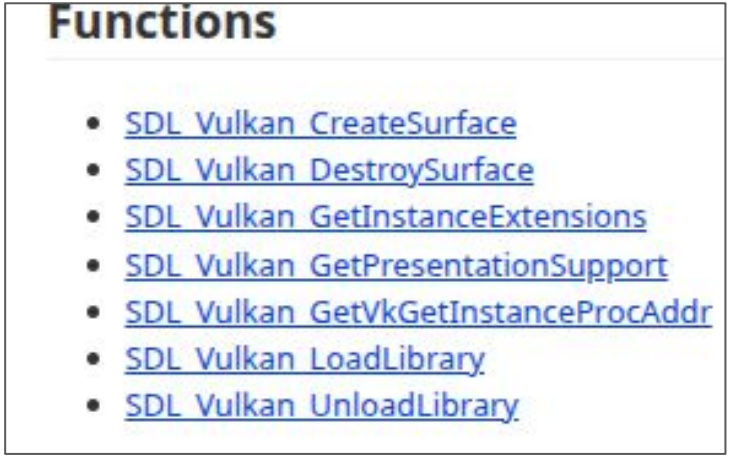
- It is fine to use C++ abstractions, but context matters. I probably would want to control the lifetime of this texture in a ‘resource manager class’
- Remember, you are managing CPU and GPU resources. Often times GPU resources can be shared. Think about the ‘flyweight design pattern’ here.

SDL 3D Graphics

New 3D Graphics API Layer

SDL3 and Graphics APIs (1/2)

- SDL3 provides a way to associate a window with other graphics APIs
  - This gives you full control over graphics rendering beyond the basic 2D graphics in SDL3 using an API (e.g. ‘Vulkan’
- See Also:
  - https://wiki.libsdl.org/SDL3/SDL_WindowFlags



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

SDL3 and Graphics APIs (2/2)

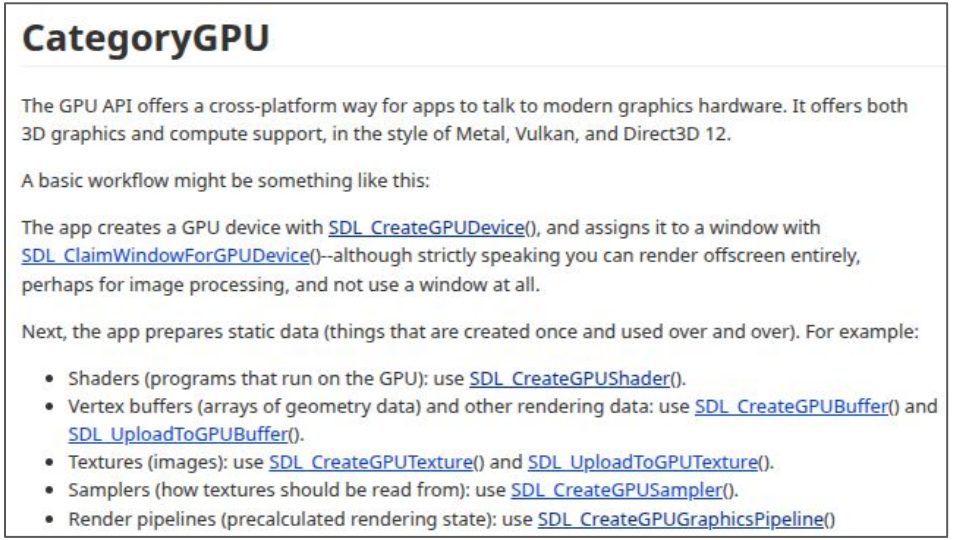
- However **-- New to SDL3** is a full **SDL_GPU API**
  - This provides an abstraction layer on top of Vulkan, Metal, D3D, etc. -- this time **you are in control**
  - SDL_GPU allows for you to write programmable GPU compute, vertex, and fragment shaders.
  - SDL_GPU provides the ‘common api’ for otherwise setting up and managing these resources.

Functions

- SDL_AcquireGPUCommandBuffer
- SDL_AcquireGPUSwapchainTexture
- SDL_BeginGPUComputePass
- SDL_BeginGPUCopyPass
- SDL_BeginGPURenderPass
- SDL_BindGPUComputePipeline
- SDL_BindGPUComputeSamplers
- SDL_BindGPUComputeStorageBuffers
- SDL_BindGPUComputeStorageTextures
- SDL_BindGPUFragmentSamplers
- SDL_BindGPUFragmentStorageBuffers
- SDL_BindGPUFragmentStorageTextures
- SDL_BindGPUGraphicsPipeline
- SDL_BindGPUIndexBuffer
- SDL_BindGPUVertexBuffers
- SDL_BindGPUVertexSamplers
- SDL_BindGPUVertexStorageBuffers
- SDL_BindGPUVertexStorageTextures
- SDL_BlitGPUTexture
- SDL_CalculateGPUTextureFormatSize
- SDL_CancelGPUCommandBuffer
- SDL_ClaimWindowForGPUDevice
- SDL_CopyGPUBufferToBuffer

SDL_GPU API - Built into SDL3

- The SDL_GPU API is an abstraction layer on top of either Vulkan/Metal/Direct3D 12 depending on your platform
- This allows you to **write graphics code once** to support most major graphics APIs
  - (Note: There is even potentially a WebGPU experiment in the community in progress as I am aware)



CategoryGPU

The GPU API offers a cross-platform way for apps to talk to modern graphics hardware. It offers both 3D graphics and compute support, in the style of Metal, Vulkan, and Direct3D 12.

A basic workflow might be something like this:

The app creates a GPU device with SDL_CreateGPUDevice(), and assigns it to a window with SDL_ClaimWindowForGPUDevice()--although strictly speaking you can render offscreen entirely, perhaps for image processing, and not use a window at all.

Next, the app prepares static data (things that are created once and used over and over). For example:

- Shaders (programs that run on the GPU): use SDL_CreateGPUShader().
- Vertex buffers (arrays of geometry data) and other rendering data: use SDL_CreateGPUBuffer() and SDL_UploadToGPUBuffer().
- Textures (images): use SDL_CreateGPUTexture() and SDL_UploadToGPUTexture().
- Samplers (how textures should be read from): use SDL_CreateGPUSampler().
- Render pipelines (precalculated rendering state): use SDL_CreateGPUGraphicsPipeline()

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

SDL_GPU API (1/4)

- The SDL_GPU API is more verbose than what we previously did -- but easier to use versus the modern graphics APIs
  - (And you get the advantage of multiple backend renderers)

```
bool done=false;
while(!done){
    SDL_Event event;
    while(SDL_PollEvent(&event)){
        if(SDL_EVENT_QUIT == event.type){
            done = true;
        }
    }

    // render
    //
    // acquire command buffer
    SDL_GPUCommandBuffer* cmdBuffer = SDL_AcquireGPUCommandBuffer(gpuDevice);
    // acquire swapchain texture
    SDL_GPUTexture* swapChainTexture;
    SDL_WaitAndAcquireGPUSwapchainTexture(cmdBuffer,window,&swapChainTexture,nullptr,nullptr);
    // begin render pass
    SDL_GPUColorTargetInfo colorTargetInfo = {
        .texture = swapChainTexture,
        .clear_color = {0.0,0.5,0.7,1.0},
        .load_op = SDL_GPU_LOADOP_CLEAR,
        .store_op = SDL_GPU_STOREOP_STORE
    };
    SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(cmdBuffer,&colorTargetInfo,1,nullptr);
    SDL_EndGPURenderPass(renderPass);
    // more render passes
    // ...
    // submit command buffer so GPU can process the commands
    if(!SDL_SubmitGPUCommandBuffer(cmdBuffer)){
        return -1;
    }

}
```

SDL_GPU API (2/4)

- All of the window creation and event handling code we learned about is the same

```
bool done=false;
while(!done){
    SDL_Event event;
    while(SDL_PollEvent(&event)){
        if(SDL_EVENT_QUIT == event.type){
            done = true;
        }
    }

    // render
    //
    // acquire command buffer
    SDL_GPUCommandBuffer* cmdBuffer = SDL_AcquireGPUCommandBuffer(gpuDevice);
    // acquire swapchain texture
    SDL_GPUTexture* swapChainTexture;
    SDL_WaitAndAcquireGPUSwapchainTexture(cmdBuffer,window,&swapChainTexture,nullptr,nullptr);
    // begin render pass
    SDL_GPUColorTargetInfo colorTargetInfo = {
        .texture = swapChainTexture,
        .clear_color = {0.0,0.5,0.7,1.0},
        .load_op = SDL_GPU_LOADOP_CLEAR,
        .store_op = SDL_GPU_STOREOP_STORE
    };
    SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(cmdBuffer,&colorTargetInfo,1,nullptr);
    SDL_EndGPURenderPass(renderPass);
    // more render passes
    // ...
    // submit command buffer so GPU can process the commands
    if(!SDL_SubmitGPUCommandBuffer(cmdBuffer)){
        return -1;
    }

}
```

SDL_GPU API (3/4)

- The ‘rendering’ portion however is what we are in control of.
- i.e.
  - There is no SDL_Renderer, we are writing our own.
  - We write our own ‘SDL_RenderTexture’ (which again, is pixel data on a quad) from scratch

```
bool done=false;
while(!done){
    SDL_Event event;
    while(SDL_PollEvent(&event)){
        if(SDL_EVENT_QUIT == event.type){
            done = true;
        }
    }

    // render
    //
    // acquire command buffer
    SDL_GPUCommandBuffer* cmdBuffer = SDL_AcquireGPUCommandBuffer(gpuDevice);
    // acquire swapchain texture
    SDL_GPUTexture* swapChainTexture;
    SDL_WaitAndAcquireGPUSwapchainTexture(cmdBuffer,window,&swapChainTexture,nullptr,nullptr);
    // begin render pass
    SDL_GPUColorTargetInfo colorTargetInfo = {
        .texture = swapChainTexture,
        .clear_color = {0.0,0.5,0.7,1.0},
        .load_op = SDL_GPU_LOADOP_CLEAR,
        .store_op = SDL_GPU_STOREOP_STORE
    };
    SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(cmdBuffer,&colorTargetInfo,1,nullptr);
    SDL_EndGPURenderPass(renderPass);
    // more render passes
    // ...
    // submit command buffer so GPU can process the commands
    if(!SDL_SubmitGPUCommandBuffer(cmdBuffer)){
        return -1;
    }

}
```

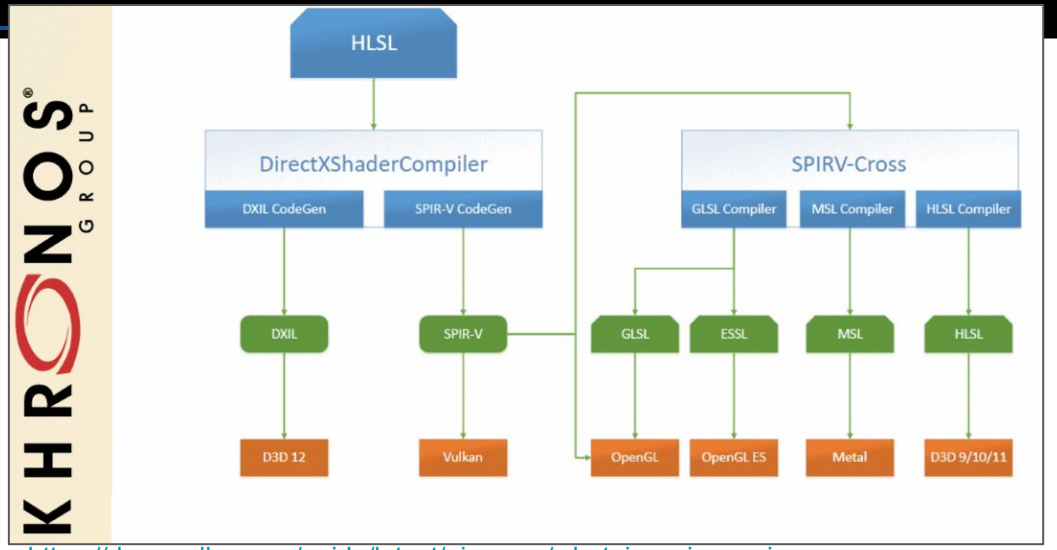
SDL_GPU API (4/4)

- For programming with Modern Graphics APIs, there are three big ideas the SDL_GPU provides an abstraction for:
  - 1. Shaders
  - 2. Buffers
  - 3. Command Buffers (and Render Passes)

```
bool done=false;
while(!done){
    SDL_Event event;
    while(SDL_PollEvent(&event)){
        if(SDL_EVENT_QUIT == event.type){
            done = true;
        }
    }

    // render
    //
    // acquire command buffer
    SDL_GPUCommandBuffer* cmdBuffer = SDL_AcquireGPUCommandBuffer(gpuDevice);
    // acquire swapchain texture
    SDL_GPUTexture* swapChainTexture;
    SDL_WaitAndAcquireGPUSwapchainTexture(cmdBuffer,window,&swapChainTexture,nullptr,nullptr);
    // begin render pass
    SDL_GPUColorTargetInfo colorTargetInfo = {
        .texture = swapChainTexture,
        .clear_color = {0.0,0.5,0.7,1.0},
        .load_op = SDL_GPU_LOADOP_CLEAR,
        .store_op = SDL_GPU_STOREOP_STORE
    };
    SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(cmdBuffer,&colorTargetInfo,1,nullptr);
    SDL_EndGPURenderPass(renderPass);
    // more render passes
    // ...
    // submit command buffer so GPU can process the commands
    if(!SDL_SubmitGPUCommandBuffer(cmdBuffer)){
        return -1;
    }

}
```

GPU API - Big Idea #1 Shaders

- 1. Shaders
  - These are tiny programs that execute on the GPU in parallel to do work
  - We use specific types of shaders to build a ‘pipeline’ to draw graphics
    - (or otherwise just perform ‘compute’)
- Shaders are compiled to a binary [Standard Portable Intermediate Representation (SPIR-V)] using a tool like glslc or shadercross (generally) ahead of time



https://docs.vulkan.org/guide/latest/_images/what_is_spirv_spriv_cross.png

```
#version 460

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec4 a_color;

layout (location = 0) out vec4 v_color;

void main()
{
    gl_Position = vec4(a_position, 1.0f);
    v_color = a_color;
}
```

An example ‘vertex shader’ program on the left in the glsl language.

GPU API - Big Idea #2 Buffers (1/3)

- Buffers are the data that ‘shaders’ operate on.
  - Buffer data must be uploaded (copied/streamed) to the GPU
- Specific buffers store different types of data
  - e.g. Vertex Buffers store vertices
  - e.g. Index Buffers store indices for geometry
  - A more generic buffer called a Shader Storage Object stores arbitrary data

SDL_CreateGPUBuffer

Creates a buffer object to be used in graphics or compute workflows.

Header File

Defined in <SDL3/SDL_gpu.h>

Syntax

```
SDL_GPUBuffer * SDL_CreateGPUBuffer(
    SDL_GPUDevice *device,
    const SDL_GPUBufferCreateInfo *createinfo);
```

Function Parameters

SDL_GPUDevice *	**device**	a GPU Context.
const SDL_GPUBufferCreateInfo *	**createinfo**	a struct describing the state of the buffer to create.

- Buffers: https://wiki.libsdl.org/SDL3/SDL_BindGPUVertexBuffers
- Uniforms: https://wiki.libsdl.org/SDL3/SDL_PushGPUVertexUniformData

GPU API - Big Idea #2 Buffers (2/3)

- Note:
  - We can ‘modify’ the buffer data per-frame by updating ‘uniform variables’
  - ‘uniforms’ are constants that we only update prior to our shaders executing.
    - Uniforms do not change while the shaders execute, but we can modify them prior to execution to do something like transform vertices in a vertex buffer

SDL_CreateGPUBuffer

Creates a buffer object to be used in graphics or compute workflows.

Header File

Defined in <SDL3/SDL_gpu.h>

Syntax

```
SDL_GPUBuffer * SDL_CreateGPUBuffer(
    SDL_GPUDevice *device,
    const SDL_GPUBufferCreateInfo *createinfo);
```

Function Parameters

SDL_GPUDevice *	**device**	a GPU Context.
const SDL_GPUBufferCreateInfo *	**createinfo**	a struct describing the state of the buffer to create.

- Buffers: https://wiki.libsdl.org/SDL3/SDL_BindGPUVertexBuffers
- Uniforms: https://wiki.libsdl.org/SDL3/SDL_PushGPUVertexUniformData

GPU API - Big Idea #2 Buffers (3/3)

- Note:
  - For completeness I'm putting an example here, but the main takeaway is there is a series of buffers we setup to control 'exactly' how / when / and how often to upload data to the GPU.

```
12 // a list of vertices to be passed into our vertex buffer
13 static Vertex vertices[]
14 {
15     {-0.5f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // top left-vertex
16     {-0.5f, -0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f},    // bottom-left
17     {0.5f,  -0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f},    // bottom-right
18
19     {0.5f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // top-right
20     {-0.5f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // top left-vertex
21     {0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f}     // bottom right vertex
22 };
```

```
// Setting up Vertex Buffer Object (VBO)
SDL_GPUBufferCreateInfo bufferInfo{};
bufferInfo.size                         = sizeof(vertices);
bufferInfo.usage                        = SDL_GPU_BUFFERUSAGE_VERTEX;
vertexBuffer                            = SDL_CreateGPUBuffer(device, &bufferInfo);

// A transfer buffer is used to setup how we upload VBO to GPU
SDL_GPUTransferBufferCreateInfo transferInfo{};
transferInfo.size               = sizeof(vertices);
transferInfo.usage              = SDL_GPU_TRANSFERBUFFERUSAGE_UPLOAD;
transferBuffer                  = SDL_CreateGPUTransferBuffer(device, &transferInfo);

// Transer buffer otherwise helps us map CPU to GPU memory
// One other step (a copy pass) otherwise completes the upload
Vertex* data = (Vertex*)SDL_MapGPUTransferBuffer(device, transferBuffer, false);
SDL_memcpy(data, (void*)vertices, sizeof(vertices));
```

GPU API - Big Idea #3 Command Buffers

- 3. Command Buffers
  - Command buffers wrap up the ‘actions’ that we want to perform and batch them
    - This is the ‘state’ needed to do something wit
  - Most commonly these might be ‘render commands’ or ‘render passes’ that tell exactly what we want our graphics card to do
    - (Where to render, what to render, and any additional state)

SDL_BeginGPURenderPass

Begins a render pass on a command buffer.

Header File

Defined in <SDL3/SDL_gpu.h>

Syntax

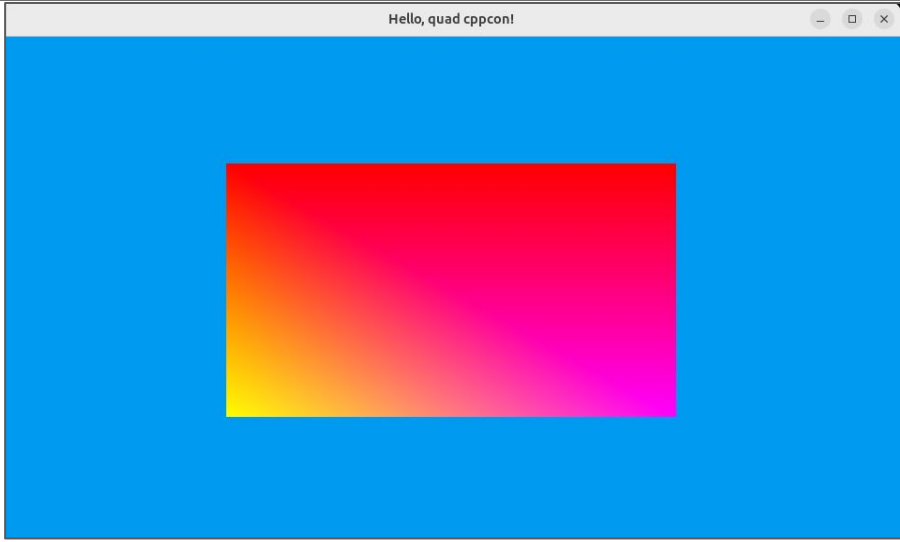
```
SDL_GPURenderPass * SDL_BeginGPURenderPass(
    SDL_GPUCommandBuffer *command_buffer,
    const SDL_GPUColorTargetInfo *color_target_infos,
    Uint32 num_color_targets,
    const SDL_GPUDepthStencilTargetInfo *depth_stencil_target_info);
```

SDL_GPU Example

- I have a completed example that will be put in the talk repository later to show you again the ‘empty’ window, but this time with ‘Vulkan’ rendering
  - ~240 lines of C++
  - 2 additional shaders (~10 lines each)
    - compiled with glslc (observe at the top)

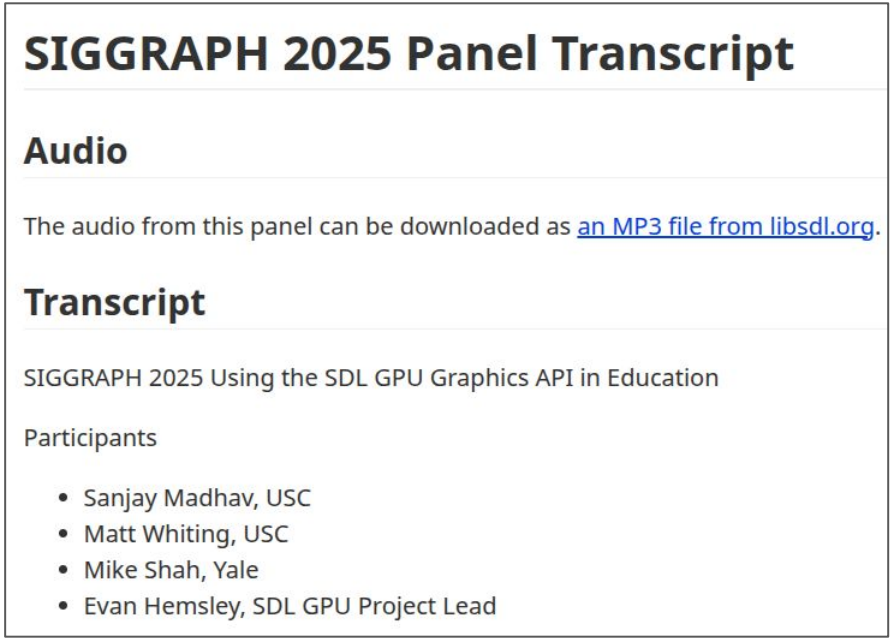
```
mike@system76-pc:~/sdl_gpu_example$ glslc -fshader-stage=vertex shaders/vertex.glsl -o shaders/vertex.spv
mike@system76-pc:~/sdl_gpu_example$ glslc -fshader-stage=fragment shaders/fragment.glsl -o shaders/fragment.spv
mike@system76-pc:~/sdl_gpu_example$ g++ source.cpp -o prog -lSDL3 && ./prog
```

```
211     // begin a render pass
212     SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(commandBuffer, &colorTargetInfo, 1, NULL);
213
214     SDL_BindGPUGraphicsPipeline(renderPass, graphicsPipeline);
215
216     SDL_GPUBufferBinding bufferBindings[1];
217     bufferBindings[0].buffer = vertexBuffer;
218     bufferBindings[0].offset = 0;
219
220     SDL_BindGPUVertexBuffers(renderPass, 0, bufferBindings, 1);
221
222     SDL_DrawGPUPrimitives(renderPass, 6, 1, 0, 0);
223
224     SDL_EndGPURenderPass(renderPass);
225     // End the render pass
```



More on SDL_GPU Design

- https://libsdl.org/siggraph2025/
- Big shoutout to Evan Hemsley who is the SDL GPU Lead
  - Shoutout to all other contributors to SDL GPU and the SDL3 project
  - (My personal interest is in teaching graphics with SDL3 in this talk as a stepping stone to the other graphics APIs)



SIGGRAPH 2025 Panel Transcript

Audio

The audio from this panel can be downloaded as an MP3 file from libsdl.org.

Transcript

SIGGRAPH 2025 Using the SDL GPU Graphics API in Education

Participants

- Sanjay Madhav, USC
- Matt Whiting, USC
- Mike Shah, Yale
- Evan Hemsley, SDL GPU Project Lead

https://libsdl.org/siggraph2025/

C++ and a Graphics Library

So what did we learn from SDL?

Will ‘`import graphics;`’ be part of C++?

- The question I want to pose is now that we have seen SDL, is there room for something like this in the standard?
  - i.e. Should ‘import graphics;’ or ‘#include <graphics>’ exist in C++?
  - Given that C++ is the dominant language for games -- this is something I and others have wondered.
- Many others have also put some proposals into action
  - (Next slide)

Selection of C++ Graphics Related Proposals

- C++ Graphics Library Proposals (Some recent examples)
  - 2020
    - 2D Graphics: A Brief Review [2020 p2005r0]
  - 2019
    - A Proposal to Add 2D Graphics Rendering and Display to C++ [2018 p0267r8] [2019 p0267r9]
  - 2018
    - Diet Graphics [2018 p1062r0]
    - Ruminations on 2D graphics in the C++ International Standard [2018 p0988r0]
    - High noon for the 2D Graphics proposal [2018 p1200r0]
    - Feedback on 2D Graphics [2018 p1225r0]
  - 2017
    - A Proposal to Add 2D Graphics Rendering and Display to C++, https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0267r3.pdf
    - Why We Should Standardize 2D Graphics for C++
    - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0669r0.pdf
  - 2016
    - INPUT DEVICES FOR 2D GRAPHICS https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0249r0.pdf
  - 2015
    - Towards improved support for games, graphics, real-time, low latency, embedded systems. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4456.pdf
    - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4526.pdf
  - 2013
    - Lightweight Drawing Library - Objectives, Requirements, Strategies
    - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3791.html
- What Belongs In The C++ Standard Library? - Bryce Adelstein Lelbach [CppNow 2021]
  - https://www.youtube.com/watch?v=OgM0MYb4DqE
- Thoughts on graphics cpp proposal
  - https://www.jeremyong.com/c++/graphics/2018/11/05/thoughts-on-the-cpp-graphics-proposal/

Note: You can search for more papers here: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/

Some Thoughts

- I would love to look at std::thread as an example success story
  - For graphics, we would have a similar implementation challenge of supporting various windowing frameworks -- but it can be done!
- Regarding graphics, I think it would be very powerful to even be able to plot a single pixel (e.g. ‘putpixel’ from Borland’s graphics.h) on a window -- all within the STL.
  - From an educators perspective, this is a dream!
  - This enables lots of interesting graphics work to be done with less wrestling of dependencies in introductory courses.
  - The graphics world does change fast, but perhaps we can provide the minimum software/hardware pixel functions to get things started.

Summary and Further Resources

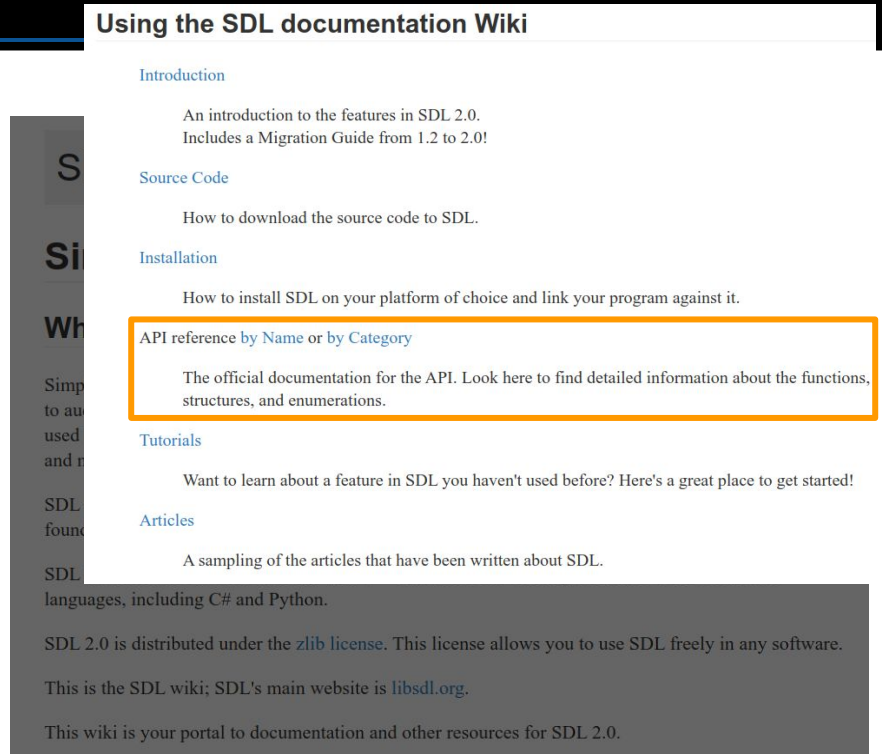
Getting help with SDL (1/3)

- The SDL wiki (link in bottom right) is probably the best resource.

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

Getting help with SDL (2/3)

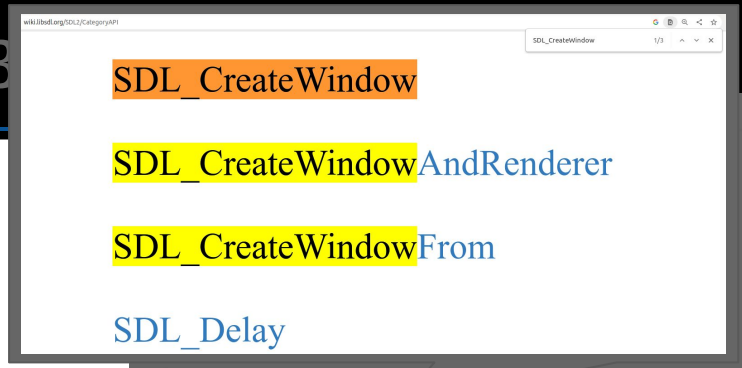
- The SDL wiki (link in bottom right) is probably the best resource.
  - Scrolling down a little bit on the page, I particularly like finding the functions by Name or Category
  - Note: There are also lots of tutorials for SDL as well.
    - Just be sure to look at version 3 for the course or potentially version 2 if that's what is setup on your machine.



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

Getting help with SDL (3

- Try now clicking ‘API reference by Name’
  - Then do ‘ctrl+f’ and type in SDL_CreateWindow
  - (Or otherwise scroll until you find the function)



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

SDL_Create Window

- Now you can observe the function, the parameters, and the return value
  - Additionally, there may be notes on how to use the function, or specific parameters
    - (i.e. often the case with flags there will be extra remarks)
- Wiki pages
  - https://wiki.libsdl.org/SDL3/SDL_CreateWindow
  - Pro tip -- take a look at similar related commands for examples or usage if you don't find a direct example.
  - Pro tip -- sometimes SDL2 examples are useful if there is not a direct SDL3 example

SDL_CreateWindow

Create a window with the specified dimensions and flags.

Header File

Defined in <SDL3/SDL_video.h>

Syntax

```
SDL_Window * SDL_CreateWindow(const char *title, int w, int h, SDL_WindowFlags flags);
```

Function Parameters

const char *	**title**	the title of the window, in UTF-8 encoding.
int	**w**	the width of the window.
int	**h**	the height of the window.
SDL_WindowFlags	**flags**	0, or one or more SDL_WindowFlags OR'd together.

Return Value

(SDL_Window *) Returns the window that was created or NULL on failure; call SDL_GetError() for more information.

Remarks

`flags` may be any of the following OR'd together:

Summary

- Today you have gotten an introduction to SDL3
  - The built-in 2D API
  - The SDL_GPU API (For when you want to take control of graphics/performance)
- Provided are some examples to get you started moving some pixels around your screen!
- SDL3 otherwise serves as a great API for also interfacing with the next generation of graphics libraries
- SDL3 serves as a potentially good example API if we want to ever have a ‘draw pixel’ type of API in C++.
  - I think it will be valuable to think more about heterogeneous compute models in the C++ language in the future.

Questions after the talk (not recorded)

- Q: I noticed OpenGL did not show up in the renderer list for the SDL_GPU example
  - A: Correct, only the modern renderers (Vulkan, D3D 12, and Metal) will show as I understand.

Thank you CppCon 2025!

Graphics Programming with SDL 3

MIKE SHAH

Web: mshah.io
YouTube www.youtube.com/c/MikeShah
Social: mikeshah.bsky.social
Courses: courses.mshah.io
Talks: http://tinyurl.com/mike-talks

60 minutes | Audience: Beginner
1:30pm - 2:30pm Fri, Sept. 19, 2025

Thank you!