SDL – Simple Direct Media Layer

A Guide by E.R. Walker

BACKGROUND

SDL is a multimedia library written in and implemented with C (with some cross-language compatibility with C++ due to the languages' shared history). For the beginner it can be best described as the C/C++ equivalent of JavaFX.

This document aims to provide an overview of the essential features of the SDL and how they can be used for implementing graphical effects, audio, user-friendly I/O and networking into one's C (or C++) applications. You will be shown the code necessary for

- Initializing SDL subsystems
- Closing SDL subsystems
- Creating SDL windows (contexts)
- Closing SDL contexts
- Creating and assigning SDL renderers to SDL contexts
- Closing SDL renderers
- Basic 2D geometry
- Using an SDL renderer to draw geometry to the context
- Load images and fonts for use in your project
- Use SDL's event subsystem for keypress mapping

It is assumed that the user has already downloaded and set up SDL in their project.

RESOURCES

SDL is a well-documented API, here are some useful links which you may need to refer to to better understand it:

1. SDL Wiki: https://wiki.libsdl.org/
2. SDL events: https://wiki.libsdl.org/SDL_Event
3. SDL_ttf Wiki: https://www.libsdl.org/projects/SDL_ttf/docs/SDL_ttf.html
4. SDL_mixer Wiki: https://www.libsdl.org/projects/SDL_mixer/docs/SDL_mixer.html
5. SDL_net Wiki: https://www.libsdl.org/projects/SDL_net/docs/SDL_net.html
6. Books on SDL: https://wiki.libsdl.org/Books

REFERENCING

The reader is welcome to utilize the contents of this document for educational purposes and reference it in written/spoken material where needed.

CONTENTS

Hold CTRL and left-click on the section you want to go to.

INITIALIZING SUBSYSTEMS

All SDL applications require the **SDL.h** header file to utilize SDL's core features. SDL subsystems (e.g. audio) can be initialized (woken up) using the **SDL_Init()** function, which takes a predefined MACRO (C's equivalent to a const variable) as a parameter. This function forwards to calling **SDL_InitSubSystem()**, so the former is a convenient abstraction.

To initialize all subsystems in SDL, one may execute the following line of code:

```
SDL_Init(SDL_INIT_EVERYTHING);
```

where *SDL_INIT_EVERYTHING* is a macro corresponding to the memory address of all subsystems. Specific subsystems can be initialized with this function using their corresponding macros, which may be chained together with the bitwise **OR** operator "|" (the pipe symbol).

Example:

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

This code initializes three different SDL subsystems – video, audio and events (inputs). Below is a list of all macros corresponding to the subsystems of the SDL:

- SDL_INIT_EVERYTHING [all subsystems]
- SDL_INIT_AUDIO [audio only]
- SDL_INIT_VIDEO [video only]
- SDL_INIT_EVENTS [events only]
- SDL_INIT_TIMER [timer only]
- SDL_INIT_JOYSTICK [joystick + events]
- SDL_INIT_GAMECONTROLLER [game controllers + joystick + events]
- SDL_INIT_HAPTIC [physical feedback (e.g. controller rumble)]
- SDL_INIT_NOPARACHUTE [ignore system compatibility]

What's the point? The SDL_Init() function can be used to check if SDL is active and ready to be used before calling functions related to it (perhaps preventing a crash!). The function itself returns a result of type **int__cdecl** (for our purposes here this can be considered a standard integer) which can be checked against zero to detect SDL is inactive:

```
if (SDL_Init(SDL_INIT_EVERYTHING) != 0) {
      SDL_Log("Could not init SDL %s\n", SDL_GetError());
      //prevent crash by returning early.
      return 1;
};
```

CLOSING SUBSYSTEMS

The function **SDL_Quit()** is called to close all initialized subsystems.

Example:

```
SDL_Init(SDL_INIT_EVERYTHING);

//use subsystems

SDL_Quit();
```

This function should be called at the end of the application's runtime as a part of the clean-up process to free heap memory allocated to SDL by the Operating System. This can be considered as SDL's primary destructor.

For quitting particular subsystems, use the **SDL_QuitSubSystem()** function, which accepts the flag of the subsystem as a parameter.

CREATING A WINDOW

After initializing SDL it can be used to generate a window (context) via the function **SDL_CreateWindow()**. The window in itself must be stored in a variable of type **SDL_Window**; because it is created as an object it must be stored specifically in a pointer to be added to heap memory.

```
SDL_Window* win = nullptr;
```

Raw pointers do not adhere to RAII (Resource Acquisition is Initialization), so it is crucial that they must always be initialized as **nullptr** to prevent nonsensical or random memory addresses being assigned.

The SDL_CreateWindow() function takes six parameters:

1. A title [const char* (C string)]
2. An initial x-pos [int]
3. An initial y-pos [int]
4. A width [int]
5. A height [int]
6. A flag [macro]*

*this flag determines how the window is presented during runtime, they may also be chained with the bitwise **OR** operator:

- SDL_WINDOW_SHOWN [created as specified, no formatting]
- SDL_WINDOW_FULLSCREEN [created fullscreen]
- SDL_WINDOW_FULLSCREENDESKTOP [created fullscreen (F11)]
- SDL_WINDOW_OPENGL [compatible with OpenGL API]
- SDL_WINDOW_VULKAN [compatible with Vulkan API]
- SDL_WINDOW_HIDDEN [window is hidden]
- SDL_WINDOW_BORDERLESS [has no taskbar]
- SDL_WINDOW_RESIZABLE [can change size]
- SDL_WINDOW_MAXIMIZED , _MINIMIZED [self-explanatory]
- SDL_WINDOW_INPUT_GRABBED [create window with input focus]
- SDL_WINDOW_ALLOW_HIGHDPI [create window with high DPI]

SDL_CreateWindow() can be used in an assignment statement for a window named “Star Wars” of size 800 x 600 as follows:

```
win = SDL_CreateWindow(
      “Star Wars”,
      0,
      0,
      800,
      600,
      SDL_WINDOW_SHOWN
);
```

PROLONGING A WINDOW

Once SDL_CreateWindow() is called the window it creates will only appear briefly (only on the frame it was created). The lifetime of a window can be extended by one of two ways:

1. The function **SDL_Delay()** which takes a integer corresponding to a delay in milliseconds, such that 1000 milliseconds = 1 second.

   Example:

```
win = SDL_CreateWindow(
      “Star Wars”,
      0,
      0,
      800,
      600,
      SDL_WINDOW_SHOWN
);

//delay for 1 second
SDL_Delay(1000);
```

   This does not accommodate a particularly interactive experience with the window, however. What SDL_Delay() does is stop the SDL thread of the program during runtime until the set time delay expires.

   Great for debugging, poor for interaction.

2. A game loop – a while loop with an input-based end condition. All game loops adhere to the following structure:
   a. Input [take user input and map it to output commands]
   b. Update [use the output to update game variables since the last update cycle]
   c. Render [translate the updated variables into graphical output]

A simple game loop may be created as follows:

```
bool isRunning = true;
SDL_Window* win = SDL_CreateWindow(
      “Game Window”,
      0,
      0,
      800,
      600,
      SDL_WINDOW_MAXIMIZED
);

//enter game loop
while(isRunning){
      //check for input (e.g. SDL events)
      //update variables (e.g. position)
      //render to screen (e.g. using SDL renderer)
};

exit(0);
```

This would create a maximized window titled “Game Window” which will last until the game loop ends (when isRunning == false). The Boolean variable may be set to false via an SDL input event (e.g. a keypress).

CLOSING A WINDOW

Even after an SDL window disappears it may still be active in heap memory (it’s a raw pointer!), which means we need to delete it manually and explicitly by calling the **SDL_DestroyWindow()** function. This function takes a single parameter – the window instance – as a means of specifying which window it is we need to have deleted.

Example:

```
win = SDL_CreateWindow(
      “Star Wars”,
      0,
      0,
      800,
      600,
      SDL_WINDOW_SHOWN
);

//window has served purpose, remove from memory.
SDL_DestroyWindow(win);
```

CREATING A RENDERER

A renderer is a graphical component which utilizes a window (context) as a canvas for drawing geometries to the screen. An SDL renderer can be created in a similar manner to an SDL window; it is stored in a variable and assigned a value via a creation function.

An SDL renderer is stored in a pointer to a variable of type **SDL_Renderer**, and assigned an instance with **SDL_CreateRenderer()**, which takes three parameters;

1. A SDL window instance [SDL_Window*]
2. A driver index [int]*
3. A flag [macro]**

*the driver index '-1' corresponds to the first driver on the system which supports the specified flags

**the flags which a renderer can be formatted with are:

- SDL_RENDERER_ACCELERATED [renderer uses hardware for performance]
- SDL_RENDERER_SOFTWARE [renderer set as a software fallback]
- SDL_RENDERER_PRESENTVSYNC [present renderer state matches refresh rate]
- SDL_RENDERER_TARGETTEXTURE [sets the renderer to draw to an SDL texture]

A renderer can be created hence:

```
SDL_Renderer* rdr = SDL_CreateRenderer(
                        window,
                        -1
                        SDL_RENDERER_ACCELERATED
                    );
```

USING A RENDERER (POINTS AND LINES)

A renderer is used to draw geometry to the screen within the boundaries of its corresponding SDL window instance. Drawing a single point (a vertex, often the size of a single pixel) on the screen within the boundaries of that window using the renderer can be achieved by using the **SDL_RenderDrawPoint()** function, which takes three parameters – as you will see for all the draw functions, the first parameter is always the renderer instance;

1. A renderer [SDL_Renderer*]
2. x-pos [float]
3. y-pos [float]

The x/y positions of the point can be abstracted to a single **SDL_Point** instance:

```
SDL_Point* point = new SDL_Point(x, y);

//or

SDL_Point point = {x, y};
```

Using the renderer to draw a point to the screen can be done as follows:

```
SDL_RenderDrawPoint(
      renderer,
      point.x,
      point.y
);
```

The result will be a single vertex drawn to the screen at x = point.x and y = point.y. Alternatively these x/y positions can be hard-coded by providing magic numbers (e.g. 2, 9, etc) in place of the point.x and point.y attributes.

Drawing lines follows a similar process with the **SDL_RenderDrawLine()** function, which takes a larger five parameters:

1. A renderer [SDL_Renderer*]
2. An x-pos start [float]
3. A y-pos start [float]
4. An x-pos end [float]
5. A y-pos end [float]

SDL knows how to draw a series of points from a starting point to an end point, which may be predefined as SDL_Point instances:

```
SDL_Point start   = {x1, y1};
SDL_Point end     = {x2, y2};
```

Where they may be used to specify the start and end of a line as follows:

The result is a straight line drawn drawn from a start to an end point.

```
SDL_RenderDrawLine(
      renderer,
      start.x,
      start.y,
      end.x,
      end.y
);
```

Multiple points and lines can be draw using the **SDL_RenderDrawPoints()** and **SDL_RenderDrawLines()** functions (respectively). SDL_RenderDrawPoints() takes three parameters like its singular counterpart, but rather than taking an x/y position it takes an array of individual points as its second parameter, then an integer specifying the number of points to be drawn from that array:

1. A renderer [SDL_Renderer*]
2. An array of points [SDL_Point[]]
3. Point count [int]

SDL_RenderDrawLines() takes the same parameters.

With the right mathematical structures, a variety of different 2D and 3D shapes can be composed using these essential building blocks.

USING A RENDERER (SPRITES)

You may want to place sprites and associated graphical assets into your application using SDL. A sprite is a 2D image (specifically a bitmap) stored as a .jpg or .png file on the operating system. SDL stores information regarding these sprites within variables of type **SDL_Surface**. These surfaces are then referenced by **textures**, which are objects of type **SDL_Texture** that correspond to SDL's interpretation of a 2D sprite.

Creating an SDL_Surface to store a bitmap can be performed as follows using the function **IMG_Load()**, which takes a single C string parameter corresponding to the image/bitmap file's location in storage on the given device – for Visual Studio the build directory is the one checked by default:

- Image file directory [const char*]

```
SDL_Surface* surface = IMG_Load("directory/file.format");
```

This surface instance can now be used by a texture instance to generate an SDL-compatible sprite using the **SDL_CreateTextureFromSurface()** function, which takes two parameters:

1. A renderer [SDL_Renderer*]
2. A surface [SDL_Surface*]

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

Now that a texture has been created SDL can render it to the screen with one of two near-identical functions; **SDL_RenderCopy()** and **SDL_RenderCopyEx()**, the latter being the preferable option for its ability to set the rotation and flip setting of the texture. SDL_RenderCopyEx() effectively means 'SDL_RenderCopy() extra options' and takes seven parameters:

1. A renderer [SDL_Renderer*]
2. A texture [SDL_Texture*]
3. A texture source rectangle* [SDL_Rect]
4. A texture distance rectangle** [SDL_Rect]
5. A rotation angle [float]
6. A rotation point [SDL_Point*]
7. A flag*** [macro]

*the source rectangle (often abbreviated to 'src') determines how much of the texture is drawn, this can be replaced with **NULL** to draw the entire texture.

the destination rectangle (often abbreviate to 'dst') determines the rendering area of the texture, which affects how the texture is stretched, this can be replaced with **NULL to fill the entire rendering target with the texture.

***flags determine the flip setting of the texture, there are three – all self-explanatory:

- SDL_FLIP_HORIZONTAL
- SDL_FLIP_VERTICAL
- SDL_FLIP_NONE

Using the SDL_RenderCopyEx() function, a texture can be drawn to the screen as follows:

```
SDL_RenderCopyEx(
      renderer,
      texture,
      src,
      dst,
      0.f,
      new SDL_Point(),
      SDL_FLIP_NONE
);
```

It is good practice to destroy a texture once you no longer require it using the **SDL_DestroyTexture()** function, which takes a single parameter – the texture you want destroyed:

- A texture [SDL_Texture*]

```
SDL_DestroyTexture(texture);
```

There is no SDL_DestroySurface() function in the SDL API, so the either **delete** keyword or the **SDL_FreeSurface()** function must be employed to release a surface pointer from memory:

```
delete surface;
```

```
SDL_FreeSurface(surface);
```

USING A RENDERER (TEXT)

Before text can be drawn to the screen with a renderer a font needs to be loaded, and before a font can be loaded SDL's TTF subsystem needs to be initialized with **TTF_Init()**. This is where we employ the **SDL_ttf.h** header to provide us with functions for doing so. Fonts are stored in TTF (TrueType Font) files (.ttf) which originated from Macintosh computer. Data regarding these fonts can be stored in variables of type **TTF_Font** and loaded using the **TTF_OpenFont()** function, which takes two parameters:

1. A file name [const char*]
2. A font size [int]

```
TTF_Font* font = TTF_OpenFont(
                        "directory/font.ttf",
                        12
                 );
```

This block of code loads a font from a specified directory and sets its font size (pixel height) to 12.

Note: It is entirely valid to provide C++ strings as parameters in place of C strings (std::string instead of const char). The C++ string class enables you to access each string instance as a C string using the member function* ***.c_str()****.*

Once a font has been loaded and assigned to a variable, it can then be assigned to a surface with the **TTF_RenderText_Blended()** function, which takes three parameters:

1. A font [SDL_Font*]
2. Text contents [const char*]
3. A color [SDL_Color]*

*an SDL_Color instance is a structure of hexadecimal values which are interpreted by the GPU as values for each color channel (R – Red, G – Green, B – Blue and A - Alpha), where the alpha channel determines the opacity of the color (how visible it is).

For example, if you want to store the color yellow in an SDL_Color instance you would need to use the following block of code (or something equivalent):

```
SDL_Color color = {0xFF, 0xFF, 0, 0xFF};
```

The '0x' is a prefix for specifying a hex value.

```
SDL_Surface* surface = TTF_RenderText_Blended(
                            font,
                            contents,
                            color
                       );
```

The text contents of a string of text may well be passed as a std::string (C++ string), fear not as the .c_str() member will help us here!

```
SDL_Surface* surface = TTF_RenderText_Blended(
                            font,
                            contents.c_str(),
                            color
                       );
```

As with any SDL_Surface we can use it to create an SDL_Texture. For a text asset we can use the SDL_CreateTextureFromSurface() function as follows:

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

To render the text texture to the screen we use the function SDL_RenderCopy(), as text seldom needs to be rotated or flipped when used for UI. When setting the 'src' and 'dst' SDL_Rect parameters for SDL_RenderCopy we may reference the function **SDL_QueryTexture()** as a means of setting the width and height parameters of the text texture. SDL_QueryTexture() takes five parameters:

1. A texture for the text [SDL_Texture*]
2. The texture's format [Uint32*] *
3. The texture's access setting [int*] *
4. A pointer to its width [int*] **
5. A pointer to its height [int*] **

**NULL may be provided to these parameters where format and access are not of concern.*

***Width and height variables may be unassigned.*

```
int width;
int height;

SDL_QueryTexture(
      texture,
      NULL,
      NULL,
      &width,
      &height
);
```

SDL_QueryTexture() may be used as follows:

```
SDL_Rect dst = {
      x,
      y,
      width,
      height
};

SDL_RenderCopy(
      renderer,
      texture,
      NULL,
      &dst
);
```

A ‘dst’ SDL_Rect may be created thereafter using the width and height and used to draw the text texture to the screen;

RENDERER UPDATE

The function **SDL_RenderPresent()** is used when updating the state of the renderer to show what it has drawn to the screen between calls. It is crucial for implementing convincing animations and it takes a single parameter – the renderer in question:

- A renderer [SDL_Renderer*]

This function is usually called after all the rendering has been performed.

CLEARING THE SCREEN

Before updating the render state it is good practice to clear the screen and prepare it for subsequent drawing by the renderer. The function **SDL_RenderClear()** is used to wipe a corresponding SDL context clean using the current *draw color* of the renderer. The function takes a single parameter – the renderer:

- A renderer [SDL_Renderer*]

```
SDL_RenderClear(renderer);
```

The draw color of the renderer determines the color in which geometries are drawn in. The draw color of a render can be set using the **SDL_SetRenderDrawColor()** function, which takes four parameters:

1. A renderer [SDL_Renderer*]
2. Red channel [float]
3. Green channel [float]
4. Blue channel [float]

and can be set as follows (e.g. for a black draw color):

```
SDL_Color color = {0, 0, 0, 0xFF};

SDL_SetRenderDrawColor(
      renderer,
      color.r,
      color.g,
      color.b
);
```

The renderer is now primed for when SDL_RenderClear() is invoked to clear the screen. Once called, the screen will be black.

After the renderer has cleared the screen its draw color can again be changed prior to re-drawing geometry.

CLOSING A RENDERER

It is now time to remove our renderer from heap memory. We call the function **SDL_DestroyRenderer()** for this, which takes a single parameter – the renderer we want to destroy. This function is near identical to SDL_DestroyWindow():

- A renderer [SDL_Renderer*]

```
SDL_DestroyRenderer(renderer);
```

USING THE EVENT SUBSYSTEM

SDL.h contains structures which enable the user to detect input from a keyboard, mouse, gamepad/controller and many more. One structure in particular is **SDL_Event**, found in **SDL_events.h**. SDL_Event is a **union** which is a structure where all members share the same address, the union itself requiring the same amount of space as its largest member. SDL_Event contains a variety of event type substructures as members

If you explore each of these structures you will see they have two variables in common both of type **Uint32** (unassigned 32-bit (4 byte) integer):

- type The type of I/O relevant to the type of event.
- timestamp The I/O interval in milliseconds (populated by SDL_GetTicks()).

Below is a list of some events which may be useful for game input, they are accompanied by their respective input types, which can be used for setting input booleans:

- SDL_CommonEvent N/A
- **Keyboard and Mouse:**
  - SDL_KeyboardEvent SDL_KEY(UP / DOWN)
  - SDL_TextEditingEvent SDL_TEXTEDITING
  - SDL_TextInputEvent SDL_TEXTINPUT
  - SDL_MouseButtonEvent SDL_MOUSEBUTTON(UP / DOWN)
  - SDL_MouseMotionEvent SDL_MOUSEMOTION
  - SDL_MouseWheelEvent SDL_MOUSEWHEEL
  - SDL_TouchFingerEvent SDL_FINGER(UP / DOWN / MOTION)
  - SDL_DropEvent SDL_DROP(BEGIN / COMPLETE / FILE / TEXT)
- **Gamepad:**
  - SDL_JoyDeviceEvent SDL_JOYDEVICE(ADDED / REMOVED)
  - SDL_JoyAxisEvent SDL_JOYAXISMOTION
  - SDL_JoyButtonEvent SDL_JOYBUTTON(UP / DOWN)
  - SDL_ControllerDeviceEvent SDL_CONTROLLERDEVICE(ADDED / REMOVED / REMAPPED)
  - SDL_ControllerAxisEvent SDL_CONTROLLERAXISMOTION
  - SDL_ControllerButtonEvent SDL_CONTROLLERBUTTON(UP / DOWN)

With this library of events at our disposal we can create an event instance as follows:

```
SDL_Event event;
```

This event instance can now be used to access keycodes for each key on a keyboard (for example), which can be used to store system input in a variable of type **SDL_Keycode**. Input from a keyboard may be accessed via this event instance as follows:

```
event.key.keysym.sym;
```

Here we're accessing the 'keysym' member of the SDL_KeyBoardEvent structure, which is of type **SDL_Keysym** and stores the virtual representation of the key that was pressed/released under a further member 'sym', of type **SDL_Keycode**. Whichever key is pressed or released during runtime – it is detected by the event instance and accessed via 'sym'. Generally speaking; any event of a particular type can be detected with the following syntax:

```
<SDL_Event var>.<type>.<code>;
```

In order for the SDL event subsystem to detect user input, the **SDL_PollEvent()** function must be applied. This function takes a single parameter:

- An event address [SDL_Event*]

Below is a code sample of how it may be used to detect events within a game loop:

```
while (is_running) {
        // input
        while (SDL_PollEvent(&event)) {
            if ((event.type == SDL_KEYDOWN || event.type == SDL_KEYUP) && event.key.repeat == 0) {
                // map input to actions and effects
        }
        // update
        // render
    }
```

LOADING AN AUDIO FILE

SDL_mixer.h provides the facilities for loading audio files, storing them in variables and playing them during an application's runtime. Before an audio file can be loaded, the SDL mixer needs to be initialized with the **Mix_OpenAudio()** function, which takes four parameters:

- The play frequency [int]
- A format flag [Uint16]
- The number of channels [int]
- The chunk size of audio [int]

A .wav audio file may be stored in a pointer of type **Mix_Chunk** after being loaded with the function **Mix_LoadWAV()**, which takes a single parameter – the directory path of the desired audio file:

- Audio file name [const char*]

PLAYING AN AUDIO FILE

Once an audio file has been loaded and assigned to a variable it can be played using the function **Mix_PlayChannel()**, which takes three parameters:

- The channel to play on [int]*
- The sound file [Mix_Chunk*]
- The number of loops [int]

*-1 can be provided for the first available channel.

Once an audio file has been used it can be freed with the **Mix_FreeChunk()** function, which takes a pointer to said audio file as a parameter.