Intro to Software Rendering with SDL2

Lets display some pixel data on the screen with SDL2 in C. There are a few ways to do this, each with their own trade-offs. I am going to assume that you already have SDL2 set up. Here is the basic framework that our program will use, with the relevant pieces slotted in where the comments are.

```c
#include <SDL2/SDL.h>

int main()
{
    SDL_Init(SDL_INIT_VIDEO);

    SDL_Window * window = SDL_CreateWindow("",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        640, 480,
        0);

    //
    // Set up the method for displaying the pixel buffer.
    //

    while (1)
    {
        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
            if (event.type == SDL_QUIT) exit(0);
        }

        //
        // Display the pixel buffer here.
        //
    }
}
```

Method 1: *Accessing the Window Buffer*

The simplest method of getting pixels on screen is the function `SDL_GetWindowSurface`. It gives back an `SDL_Surface` containing our pixel buffer. We can then modify this buffer and display the changes we made using `SDL_UpdateWindowSurface`.

```c
SDL_Surface * window_surface = SDL_GetWindowSurface(window);
```

Within this structure there is a pointer to the raw pixel buffer called `pixels`. This pointer is of type `void *`, so we need to cast it to be able to use it. You can do this by assigning the value of the internal pointer to your own pointer of the correct type — `unsigned int`[1].

```
unsigned int * pixels = window_surface->pixels;
```

You can now access the pixel buffer directly. To set values of pixels using their `(x, y)` coordinates, you must use the formula `x + y * width` — where `width` is the width of the pixel buffer — to index into the buffer.

```
// The coordinates of our pixel.
int x = 10;
int y = 30;

// You can get the width value from the surface itself if needed.
int width = window_surface->w;

pixels[x + y * width] = 0xffffffff;
```

Here I have used hexadecimal notation to specify the colour (white) that the pixel should be set. This is the part where this method of accessing the pixel buffer becomes more tricky, as the channel order (e.g. `RGBA`, `ARGB`, `BGRA`, …) can vary between platforms and for other factors. You can find out what the pixel format is by reading the value at `window_surface->format->format`, and you can print that out in a readable way using `SDL_GetPixelFormatName`.

```
puts(SDL_GetPixelFormatName(window_surface->format->format));
```

Another necessary evil of this method is that if you change resolution at any point, perhaps if you are using a resizeable window or switch to full-screen mode, you will need to call `SDL_GetWindowSurface` again and get a new surface, as the original pointer is now invalid.

```
// You should get the new pixel buffer whenever
// a SIZE_CHANGED window event comes in.
if (event.type == SDL_WINDOWEVENT)
{
    if (event.window.event == SDL_WINDOWEVENT_SIZE_CHANGED)
    {
        window_surface = SDL_GetWindowSurface(window);
        pixels = window_surface->pixels;
    }
}
```

And finally, the format of the pixels in the window is defined by the platform. SDL provides a few functions to deal with this, including `SDL_MapRGBA` to get a correctly formatted pixel, and `SDL_GetRGBA` to get the components back from a pixel. There are also versions that omit the alpha channel: `SDL_MapRGB` and `SDL_GetRGB`.

```
// Pack these RGBA values into a pixel of the correct format.
Uint32 pixel = SDL_MapRGBA(window_surface->format, 200, 130, 100, 255);
```

```
// Get the components back out of that packed pixel.
Uint8 r, g, b, a;
SDL_GetRGBA(pixel, window_surface->format, &r, &g, &b, &a);
```

If you're using a predefined colour palette, it is worth converting all of your colours once up front, then saving the packed pixels for use in your graphics code.

Summary

This method is great to get jump started with software rendering in SDL2, and may work best on some platforms, such as the Raspberry Pi. It takes a single function call to get access to the buffer, and another to update the window and show your graphics on screen. The downsides are that you may need to perform a pixel format conversion if you aren't using the same pixel format in the rest of your code, and that you will need to access a new pixel buffer whenever the resolution of the window changes. Also, there is no way to enable vertical sync.

Click here to see the complete code example for accessing the window buffer.

Method 2: *Streaming Texture*

The second method of displaying a pixel buffer on screen is via a 'streaming' texture. This is a chunk of GPU-side memory that is displayed on screen, which you can modify using a graphics API such as OpenGL or DirectX. SDL2 offers a simple method to use streaming textures that handles the API calls for you, using its 2D Accelerated Rendering API. This method requires more work to set up, but provides features that may make it easier for you to work with the pixel buffer.

Firstly, we must create an `SDL_Renderer`, and an `SDL_Texture`.

```
// Create a renderer with V-Sync enabled.
SDL_Renderer * renderer = SDL_CreateRenderer(window,
    -1, SDL_RENDERER_PRESENTVSYNC);

// Create a streaming texture of size 320 x 240.
SDL_Texture * screen_texture = SDL_CreateTexture(renderer,
    SDL_PIXELFORMAT_RGBA8888, SDL_TEXTUREACCESS_STREAMING,
    320, 240);
```

You may have noticed that in the call to `SDL_CreateTexture` that I specified the pixel format using the flag `SDL_PIXELFORMAT_RGBA8888`. This means that our texture is guaranteed to use the `RGBA` format, but there are many formats you can choose. I also specified the resolution (`320x240`) in this call, and enabled V-Sync. This resolution does not need to be the same as your window resolution, as you will see later on.

As the texture is likely to be stored in GPU memory, we cannot access it with a pointer. In order to have a buffer we can modify directly we must create it ourselves.

```
// Create a buffer large enough to hold the same
// number of pixels as the GPU-side texture.
// 320 is our width.
// 240 is our height.
// 4 is the size in bytes of a single RGBA pixel.
unsigned int * pixels = malloc(320 * 240 * 4);
```

We now have everything we need to display our graphics on screen, but this must be done in a few stages. Firstly, we must get our pixel data from the `pixels` buffer into the `screen_texture`. This is done using the function `SDL_UpdateTexture`.

```
SDL_UpdateTexture(screen_texture, NULL, pixels, 320 * 4);
```

The `NULL` in this call is where you can specify a rectangular sub-section of the texture to update, if you only want to change it partially. We want to change the whole thing, so we pass `NULL` instead of a pointer to an `SDL_Rect`. The final parameter is the 'pitch' of the buffer, which is the number of bytes that make up a single row of pixels — `320` is our width, and `4` is the number of bytes in a single pixel.

Now that our pixel data is in the texture, we need to display that texture on the screen. This is done using `SDL_RenderCopy`.

```
SDL_RenderCopy(renderer, screen_texture, NULL, NULL);
```

Same as before, the `NULL`s mean display the entire texture across the entire window; texture will be stretched to fill the entire window. This allows us to display a pixel buffer that is lower (or higher) resolution than our window. Now that we have rendered our texture we must display the result of our rendering using `SDL_RenderPresent`.

```
SDL_RenderPresent(renderer);
```

And now our internal pixel buffer should be displayed on screen.

Extras

I mentioned that our call to `SDL_RenderCopy` will perform stretching, but this stretching is done 1:1 with the window resolution, meaning the original pixel buffer may be stretched by different amounts in x and y. To change this, ensuring that the aspect ratio of the original image is sustained we can use `SDL_RenderSetLogicalSize` just after creating the renderer. It uses letterboxing to achieve this. This function will also change the mouse position values such that they line up with the 'logical' resolution specified.

```
// Set the logical resolution to the resolution
// of our internal pixel buffer.
SDL_RenderSetLogicalSize(renderer, 320, 240);
```

Scaling will now keep a fixed aspect ratio, but if you are building a program with low-resolution ('retro') graphics you may notice that each pixel on screen is not exactly the same size. This occurs when the internal (or logical) resolution is not a factor of the actual window resolution. This can also be fixed by using another SDL function.

```
// This takes a boolean value; 1 isn't a special number.
SDL_RenderSetIntegerScale(renderer, 1);
```

With these features enabled you are likely to see black bars around the pixel buffer. You may want to set the colour of this area, which you can do like so.

```
// Set the colour to red.
SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
```

Summary

This method has many features that make it easier and more reliable to work with than the first, but it does take some more work to set it up. I recommend that anyone using SDL2 that wants to perform software rendering should give both methods a try, as they differ in performance on various platforms, though this second method is more often the better choice.

Click here to see the complete code example for rendering with a streaming texture.

1. It is more correct to use `uint32_t`, which is defined in `<stdint.h>`. SDL also provides a `typedef`'d version of this type called `Uint32`, which you could use instead.↩

This page was last updated on **2022-02-05**.