

Wefx

The Wefx Devs

2022-04-08

Abstract

Wefx is a simple graphics library for drawing using C, WASM (Web Assembly), and an HTML canvas. It aims to provide an introduction to graphics programming using C and WASM. Wefx is meant to be a teaching / learning tool.

Contents

1	wefx	2
1.1	Using The Project	2
1.1.1	Compiling	4
1.1.2	Serving	4
1.1.3	Writing Code	4
1.1.4	API: Getting Started	5
1.1.5	Windows OS	5
2	Wefx Drawing	5
2.1	Opening a Canvas - wefx_open	6
2.2	Convert Integers to RGB - rgb_to_int	6
2.3	Set the Foreground Color - wefx_color	7
2.4	Draw a Single Point - wefx_point	7
2.5	Set the Background Color - wefx_clear_color	7
2.6	Clear the Screen - wefx_clear	7
2.7	Draw a Line - wefx_line	7
2.8	Draw a Circle - wefx_circle	8
2.9	Draw the Buffer to Screen	9
2.10	Get Screen Dimensions	9
3	Event Queue	9
3.1	Open Events - wefx_open_events	10
3.2	Add Events from Javascript	10
3.3	Add an Event to a Queue	11
3.4	Remove an Event from a Queue	11
4	Math Functions	12
4.1	Abs function	12
4.2	Degree To Radians	12
4.3	Radians To Degrees	12
4.4	Power	13
4.5	Floor	13

4.6	Ceiling	13
4.7	Round	13
4.8	Cosine function	13
4.9	Sine function	14
4.10	Seed Random	14
4.11	Random - rand()	15
5	Malloc Implementation - walloc	15
6	Export Function to Javascript	15

1 wefx

Wefx is a simple graphics library for drawing using C, WASM (Web Assembly), and an HTML canvas. It aims to serve a similar to gfx, but provide an introduction to using C and WASM. Wefx is meant to be a teaching / learning tool for C and graphics. Wefx is not using OpenGL / WebGL or anything like that. It is doing very basic pixel manipulation and has very simple functions to draw pixels and lines.

If you would like to see an overview of Wefx, you can watch a video of the project on YouTube

You can also download the documentation

1.1 Using The Project

The flow of the project has two steps: the build step, and the serve step:

```
| ./src + ./examples | clang | ./build/wefx.wasm |
```

```

/-----/
/

```

```
| ./build/ | web server | browser | | you! |
```

In other words, you compile the C code into a WASM, and then serve the `build` directory using a web server. You can then open a web browser and visit `http://localhost:8000` to view the running C code.

You'll need the following programs installed:

- clang
- make (optional - MacOS and Linux)
- (optional) python3

On MacOS or Linux these tools should be available already, or easily installed with homebrew (`brew install`), or Apt (`apt install`), or your local package manager.

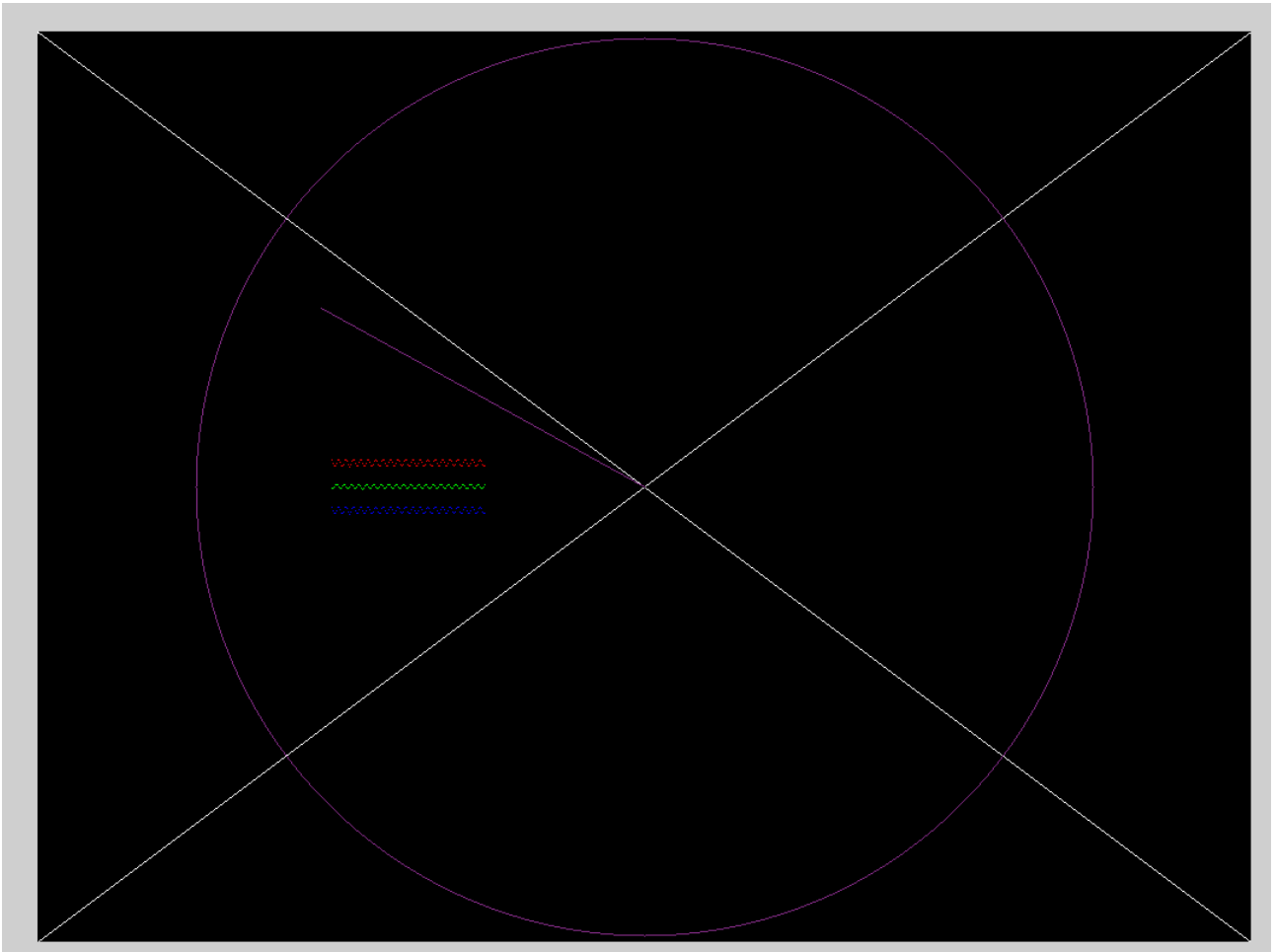


Figure 1: Example Screenshot

1.1.1 Compiling

To understand what is happening (or if you do not want to use *make*), open the *Makefile* file and look at the *build* task. There you can see how *clang* is used.

If you have *make* available, type:

```
make build
```

on the command line. This will, assuming there are no errors, create the file `./build/wefx.wasm`. Once this builds you can serve your creation by doing the following...

1.1.2 Serving

The gist of this is you need to serve the contents of the `/build` directory in a web server. You need to do this because the file that loads the newly create **wasm** file (`index.html`) can only load the **wasm** file over `http`. You can not simply open the `index.html` file directly from your file system (this is just how **wasm** loading works).

If you try to open the `index.html` file directly you will get an error like:

Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at

A basic `http` server comes with `python3`, and the `make` file will run that server if you run:

```
make serve
```

and the `python3` web server will serve the files in the *build* directory. You can then use your favorite browser and browse to `http://localhost:8000` to see the compiled code.

Note `make serve` will both recompile your code and run the web server.

If you already have a favorite server (for example I use `busyboy`), you can use that serve to serve the `build` directory instead, and then run the `make build` command to replace the **wasm** file as you play around.

For example in one shell I run:

```
busyboy --root=./build
```

Then, after I make changes to the C code, I run

```
make build
```

And then simply refresh the browser to see changes.

1.1.3 Writing Code

If just teaching / learning about graphics, you'll only need to edit the `./examples/example0.c` file. There are two entry points into that file:

Function	Usage
<code>init()</code>	Called once at the start of the app
<code>main_loop(time)</code>	Called every frame with <code>time</code> being time since app start

You can also add your own files entry files in the examples directory, and then pass them to the build script using the *MAIN* variable. For example:

```
make build MAIN=examples/example1.c
```

will build the WASM file using `example1.c` as the entry point.

1.1.4 API: Getting Started

The API calls try to emulate gfx as much as possible. Here are a few currently supported functions (see the documentation for a full reference):

Function	Does
<code>wefx_open(width, height, title)</code>	Create a canvas on which to draw
<code>wefx_clear_color(red, green, blue)</code>	Set the background color
<code>wefx_color(red, green, blue)</code>	Set the drawing color
<code>wefx_point(x, y)</code>	Draw a single point
<code>wefx_line(x1, y1, x2, y2)</code>	Draw a line from (x1,y1) to (x2,y2)
<code>wefx_clear()</code>	Clear the canvas using the background color

1.1.5 Windows OS

I have not run this on Windows, but you should be able to build it with Visual Studio. You will have to install `clang` as an add on, and then setup the build flags to the ones shown in the `Makefile`.

2 Wefx Drawing

A simple graphics library made in the spirit of gfx

To start out, we import our *wefx.h* header file. This file contains a few structs, and enumerations we will be using throughout this implementation file.

```
#include "wefx.h"
```

Additionally, we include our *wasm.h* file which will allow us to *EXPORT* functions and variables to Javascript.

```
#include "wasm.h"
```

```
typedef unsigned int color;
```

Make variables for our double buffered screen, and export the *screen* variable so Javascript can use it's contents to create an image.

```
EXPORT unsigned int *screen;  
static unsigned int *buffer;
```

We also defined some global variables for foreground and background colour, as well as a global width (*w*) and height (*h*) variable.

```
static color fg_color = 0;  
static color bg_color = 0;
```

```
static int w = 0;
static int h = 0;
```

Lastly, we reserve a spot for an event queue where we will store user events from the browser. E.g. Mouse down, mouse move, key down, etc.

```
wefx_event_queue *wefx_q = NULL;
```

2.1 Opening a Canvas - wefx_open

Here we emulate opening a window. This allocates memory for our screen buffer.

```
int wefx_open(unsigned int width, unsigned int height, const char *title)
{
    w = width;
    h = height;
    buffer = malloc(w * h * sizeof(int));
    if (buffer == NULL)
    {
        return 1;
    }
    return 0;
}
```

2.2 Convert Integers to RGB - rgb_to_int

Since we have a few functions that deal with colors, we create an internal function that will help convert three RGB values into a single integer value. This value will go directly into the screen buffer to represent a pixel color.

When we are given three values to represent RGB, we shift the colors up into a single integer. Note we need to ensure the values are not greater than 255 since we are only supporting RGB value from within that range.

In the end, the integer will look like the following (in hexadecimal):

alpha	blue	green	red
0xFF	FF	FF	FF

from within a single integer.

```
static int rgb_to_int(unsigned int red, unsigned int green, unsigned int blue)
{
    if (red > 255)
        red = 255;
    if (green > 255)
        green = 255;
    if (blue > 255)
        blue = 255;
    int color = (0xFF << 24) + (blue << 16) + (green << 8) + (red);
    return color;
}
```

2.3 Set the Foreground Color - `wefx_color`

This function simply sets our *fg_color* variable to the specified color.

Subsequent calls to draw will use this color until it is changed.

```
void wefx_color(unsigned int red, unsigned int green, unsigned int blue)
{
    fg_color = rgb_to_int(red, green, blue);
}
```

2.4 Draw a Single Point - `wefx_point`

This function sets one pixel value to a color. It set one of the values in our *buffer* array to the current value stored *fg_color*.

By setting the value at $x+y*w$ we are drawing a point at (x,y) on the screen.

```
void wefx_point(int x, int y)
{
    buffer[x + y * w] = fg_color;
}
```

2.5 Set the Background Color - `wefx_clear_color`

This color will be used as the background color for the image. Call this to set the background color, and then by calling *wefx_clear*, you can fill the entire screen buffer.

```
void wefx_clear_color(unsigned int red, unsigned int green, unsigned int blue)
{
    bg_color = rgb_to_int(red, green, blue);
}
```

2.6 Clear the Screen - `wefx_clear`

This function will “clear the screen”. What it actually does is fill the whole screen buffer with the current background color (see *wefx_clear_color*).

This is often called at the top of the render loop to reset to a blank slate before doing any drawing.

```
void wefx_clear()
{
    for (int q = 0; q < w * h; q++)
        buffer[q] = bg_color;
}
```

2.7 Draw a Line - `wefx_line`

Here we define a simple function to draw a line. It will draw from $(x1,y1)$ to $(x2,y2)$ using Bresenham’s line algorithm and the currently set foreground color (‘Bresenham’s Line Algorithm’ 2022).

```

void wefx_line(int x0, int y0, int x1, int y1)
{
    int dx = abs(x1 - x0);
    int sx = x0 < x1 ? 1 : -1;
    int dy = -abs(y1 - y0);
    int sy = y0 < y1 ? 1 : -1;
    int error = dx + dy;

    for (;;)
    {
        wefx_point(x0, y0);
        if (x0 == x1 && y0 == y1)
            break;
        int e2 = 2 * error;
        if (e2 >= dy)
        {
            if (x0 == x1)
                break;
            error = error + dy;
            x0 = x0 + sx;
        }
        if (e2 <= dx)
        {
            if (y0 == y1)
                break;
            error = error + dx;
            y0 = y0 + sy;
        }
    }
}

```

2.8 Draw a Circle - wefx_circle

This function can be called to draw a circle. It also uses the currently set foreground color. I uses the Midpoint Circle Algorithm ([‘Midpoint Circle Algorithm’ 2022](#)).

```

void wefx_circle(int x0, int y0, int r0)
{
    int x = r0;
    int y = 0;
    int err = 0;
    while (x >= y)
    {
        wefx_point(x0 + x, y0 + y);

        wefx_point(x0 + y, y0 + x);
        wefx_point(x0 - y, y0 + x);
        wefx_point(x0 - x, y0 + y);
        wefx_point(x0 - x, y0 - y);
        wefx_point(x0 - y, y0 - x);
    }
}

```



```

    wefx_point(x0 + y, y0 - x);
    wefx_point(x0 + x, y0 - y);

    y += 1;
    err += 2 * y + 1;
    if (err > 0)
    {
        x -= 1;
        err -= 2 * x + 1;
    }
}
}

```

2.9 Draw the Buffer to Screen

When we call any of the wefx draw functions, we are actually changing the pixels in a buffer. They pixels we are setting are not actually drawn to the screen.

This method is called from Javascript and asks us to draw our buffer to what it considers to be the screen.

Note: there might be a faster / better way to do this.

```

EXPORT void wefx_draw(unsigned int *screen)
{
    for (int q = 0; q < w * h; q++)
        screen[q] = buffer[q];
}

```

2.10 Get Screen Dimensions

Javascript will need to know what size we have defined our screen size to be. These methods are exposed to Javascript to get the X and Y dimensions of the buffer / screen.

```

EXPORT int wefx_xsize()
{
    return w;
}
EXPORT int wefx_ysize()
{
    return h;
}

```

3 Event Queue

In order to process browser events (keyboard and mouse input), we use a simple Queue. The Javascript that hosts this code will capture events in the browser and pass them into C using the queue. We can then use the queue to look at and process those events.

3.1 Open Events - `wefx__open_events`

Similar to how `wefx__open` created screen memory, the `wefx__open_events` function allocates memory for the event queue.

```
wefx_event_queue *wefx__open_events()
{
    wefx_q = malloc(sizeof(struct wefx_event_queue));
    if (wefx_q != NULL)
    {
        wefx_init_queue(wefx_q);
        return wefx_q;
    }

    return NULL;
}
```

When we create a new queue, we want to set the head and tail to null to mark it as empty. This is not strictly necessary, but it will make knowing if the queue is empty a bit easier.

```
void wefx_init_queue(wefx_event_queue *q)
{
    q->head = NULL;
    q->tail = NULL;
}
```

3.2 Add Events from Javascript

Javascript will call this function directly to register that an event has occurred.

Here we just take the parameters passed in from Javascript, put them into the `wefx_event` struct, and add it to the end of the queue.

```
EXPORT void wefx_add_queue_event(int type, int button, int timestamp, int key, int x, int y)
{
    // if we don't care about events drop everything
    if (wefx_q == NULL)
        return;

    wefx_event *e = malloc(sizeof(struct wefx_event));
    if (e == NULL)
    {
        // we couldn't create memory for some reason
        // this seems to happen a bit when running in
        // wasm (or maybe walloc)
        return;
    }
    e->type = type;
    e->button = button;
    e->timestamp = timestamp;
    e->key = (char)key;
    e->x = x;
    e->y = y;
}
```

```

    wefx_enqueue(wefx_q, e);
}

```

3.3 Add an Event to a Queue

wefx_enqueue takes an event and adds it to the given queue. It does a number of checks to make sure the queue is in a valid state, and adds the event if everything is ok.

```

int wefx_enqueue(wefx_event_queue *q, wefx_event *event)
{
    // create a new node to store the event
    wefx_event_node *node = malloc(sizeof(struct wefx_event_node));
    if (node == NULL)
    {
        return -1;
    }

    node->event = event;
    node->next = NULL;

    // if the queue has a tail, add us as the next behind the tail
    if (q->tail != NULL)
    {
        q->tail->next = node;
    }
    // now really make us the last node
    q->tail = node;

    // if we are first in line, go to the head
    if (q->head == NULL)
    {
        q->head = node;
    }

    return 1;
}

```

3.4 Remove an Event from a Queue

Take the first event in the queue off the queue, and return it.

```

wefx_event *wefx_dequeue(wefx_event_queue *q)
{
    if (q == NULL)
        return NULL;

    if (q->head == NULL)
    {
        return NULL;
    }
}

```

```

    wefx_event_node *n = q->head;
    wefx_event *e = n->event;
    q->head = q->head->next;
    if (q->head == NULL)
    {
        q->tail = NULL;
    }

    free(n);
    return e;
}

```

4 Math Functions

First we include the header file which define some constants and macros.

```
#include "math.h"
```

Params for the random number generator (lcg below)

```
#define RND_2_31_MINUS_ONE 2147483647
#define RND_PARK_MILLER 16807
```

4.1 Abs function

The *abs* function will return the absolute value of the given number.

```

float abs(float n)
{
    if (n < 0)
        return n * -1;
    return n;
}

```

4.2 Degree To Radians

The *to_radian* function converts a degree value to a radian value. This is simply:

$$r = d \cdot (\pi/180)$$

```

float to_radian(float degree)
{
    return degree * (M_PI / 180);
}

```

4.3 Radians To Degrees

The *to_degree* function converts a radian value to a degree value. This is simply:

$$d = r \cdot (180/\pi)$$

```
float to_degree(float radian)
{
    return radian * (180 / M_PI);
}
```

4.4 Power

```
float pow(float x, float y)
{
    float rtn = x;
    while (y > 1)
    {
        rtn *= x;
        y--;
    }
    return rtn;
}
```

4.5 Floor

```
float floor(float n)
{
    float base = (float)(int)(n - ((int)n % 1));
    if (n < 0)
        return base - 1;
    return base;
}
```

4.6 Ceiling

```
float ceil(float n)
{
    float base = (float)(int)(n - ((int)n % 1));
    if (n > 0)
        return base + 1;
    return base;
}
```

4.7 Round

```
float round(float x)
{
    float m = x - (int)x;
    if (m < 0.5)
        return floor(x);
    return ceil(x);
}
```

4.8 Cosine function

This function produces a cosine approximation. It takes a simple

First we define the curve we're going to use to emulate cosine:

$$y = x \cdot 16 \cdot (|x| - w)$$

This curve looks close to cosine, but it does not repeat. So we recursively split a line and tilt it so the our cosine-like curve starts and stops at about the right place.

$$p = \frac{x}{\pi^2}$$

$$r = p - 0.25 - \lfloor p + 0.25 \rfloor$$

now put it together and make sure the ends touch:

$$y = r \times 16 \times |r - 0.5|$$

You can play more with the way this works on the Desmos website (Rohan, n.d.)

```
float cos(float x)
{
    float p = x / M_PI2;
    float r = p - .25 - floor(p + .25);
    float y = r * 16 * (abs(r) - 0.5);
    return y;
}
```

4.9 Sine function

The sine function is just cos shifted on the x axis

$$\sin(x) = \cos\left(x - \frac{\pi}{2}\right)$$

```
float sin(float x)
{
    return cos(x - M_PID2);
}
```

4.10 Seed Random

Here we can seed the random number generator, or just use the default seed if one isn't provided. The default seed has no special meaning or value, it was just chosen at random.

```
static unsigned int SEED = 9035768;
void srand(unsigned int seed)
{
    SEED = seed;
}
```

4.11 Random - rand()

Basic random number generator. Linear congruential generator

If the increment parameter is set to zero $inc = 0$, the generator is a *multiplicative congruential generator* (MCG). If the *inc* parameter is not 0, the method is called a mixed congruential generator (Hörmann and Derflinger, n.d.).

By default, when you call *rand()* the increment value is set to zero.

```
static int lcg(int md, int mult, int inc, int seed)
{
    SEED = (mult * seed + inc) % md;
    return SEED;
}

int rand()
{
    return lcg(RND_2_31_MINUS_ONE, RND_PARK_MILLER, 0, SEED);
}
```

5 Malloc Implementation - walloc

walloc, written by S.L. Igalia, is a bare-bones implementation of malloc for use by C programs when targetting WebAssembly. It is a single-file implementation of *malloc* and *free* with no dependencies (Wingo 2022).

Here we are just defining the interface to the walloc code.

```
#ifndef WEFX_WALLOC__H
#define WEFX_WALLOC__H

void *malloc(unsigned long);
void free(void *);

#endif /* MALLOC__H */
```

6 Export Function to Javascript

Here we simply define a macro that will let the compiler know that a function should be made available in Javascript.

Prefix functions with the *EXPORT* keyword if you would like to be able to call a C function from within Javascript.

```
#ifndef WEFX_WASM_H
#define WEFX_WASM_H

#include "math.h"
#include "walloc.h"

#define EXPORT __attribute__((visibility("default")))
```

#endif

‘Bresenham’s Line Algorithm.’ 2022. *Wikipedia*, February.

Hörmann, Wolfgang, and Gerhard Derflinger. n.d. ‘A Portable Uniform Random Number Generator Well Suited for the Rejection Method,’ 6.

‘Midpoint Circle Algorithm.’ 2022. *Wikipedia*, February.

Rohan, Robert. n.d. ‘Handmade Cosine.’ *Desmos*. <https://www.desmos.com/calculator/kqdnljukzx>.

Wingo, Andy. 2022. ‘Walloc.’