Drawing to canvas in Emscripten

- Canvas via Embind
  - Choosing the canvas element
- OpenGL and SDL2
  - Drawing a rectangle
  - Integrating an event loop
    - Unblocking event loop with Asyncify
    - Unblocking event loop with "main loop" APIs
    - Handling user interactions
  - Drawing other shapes with SDL2_gfx

Different operating systems have different APIs for drawing graphics. The differences become even more confusing when writing a cross-platform code, or porting graphics from one system to another, including when porting native code to WebAssembly.

In this post you will learn a couple of methods for drawing 2D graphics to the canvas element on the web from C or C++ code compiled with Emscripten.

Canvas via Embind

If you're starting a new project rather than trying to port an existing one, it might be easiest to use the HTML Canvas API via Emscripten's binding system Embind. Embind allows you to operate directly on arbitrary JavaScript values.

To understand how to use Embind, first take a look at the following example from MDN that finds a <canvas> element, and draws some shapes on it

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
```

```
ctx.fillStyle = 'green';
ctx.fillRect(10, 10, 150, 100);
```

Here's how it can be transliterated to C++ with Embind:

```
#include <emscripten/val.h>

using emscripten::val;

// Use thread_local when you want to retrieve & cache a global JS variable once per thread.
thread_local const val document = val::global("document");

// …

int main() {
  val canvas = document.call<val>("getElementById", "canvas");
  val ctx = canvas.call<val>("getContext", "2d");
  ctx.set("fillStyle", "green");
  ctx.call<void>("fillRect", 10, 10, 150, 100);
}
```

When linking this code, make sure to pass `--bind` to enable Embind:

```
emcc --bind example.cpp -o example.html
```

Then you can serve the compiled assets with a static server and load the example in a browser:

Emscripten-generated HTML page showing a green rectangle on a black canvas.

Choosing the canvas element

When using the Emscripten-generated HTML shell with the preceding shell command, the canvas is included and set up for you. It makes it easier to build simple demos and examples, but in larger applications you'd want to include the Emscripten-generated JavaScript and WebAssembly in an HTML page of your own design.

The generated JavaScript code expects to find the canvas element stored in the `Module.canvas` property. Like other Module properties, it can be set during initialization.

If you're using ES6 mode (setting output to a path with an extension `.mjs` or using the `-s EXPORT_ES6` setting), you can pass the canvas like this:

```
import initModule from './emscripten-generated.mjs';

const Module = await initModule({
  canvas: document.getElementById('my-canvas')
});
```

If you're using regular script output, you need to declare the `Module` object before loading the Emscripten-generated JavaScript file:

```
<script>
var Module = {
  canvas: document.getElementById('my-canvas')
};
</script>
<script src="emscripten-generated.js"></script>
```

OpenGL and SDL2

OpenGL is a popular cross-platform API for computer graphics. When used in Emscripten, it will take care of converting the supported subset of OpenGL operations to WebGL. If your application relies on features supported in OpenGL ES 2.0 or 3.0, but not in WebGL, Emscripten can take care of emulating those too, but you need to opt-in via the corresponding settings.

You can use OpenGL either directly or via higher-level 2D and 3D graphics libraries. A couple of those have been ported to the web with Emscripten. In this post, I'm focusing on 2D graphics, and for that SDL2 is currently the preferred library because it's been well-tested and supports the Emscripten backend officially upstream.

Drawing a rectangle

"About SDL" section on the official website says:

> Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D.

All those features - controlling audio, keyboard, mouse and graphics - have been ported and work with Emscripten on the web too so you can port entire games built with SDL2 without much hassle.

If you're porting an existing project, check out the "Integrating with a build system" section of Emscripten docs.

For simplicity, in this post I'll focus on a single-file case and translate the earlier rectangle example to SDL2:

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

int main() {
  // Initialize SDL graphics subsystem.
  SDL_Init(SDL_INIT_VIDEO);

  // Initialize a 300x300 window and a renderer.
  SDL_Window *window;
  SDL_Renderer *renderer;
  SDL_CreateWindowAndRenderer(300, 300, 0, &window, &renderer);

  // Set a color for drawing matching the earlier `ctx.fillStyle = "green"`.
  SDL_SetRenderDrawColor(renderer, /* RGBA: green */ 0x00, 0x80, 0x00, 0xFF);
  // Create and draw a rectangle like in the earlier `ctx.fillRect()`.
  SDL_Rect rect = {.x = 10, .y = 10, .w = 150, .h = 100};
  SDL_RenderFillRect(renderer, &rect);

  // Render everything from a buffer to the actual screen.
  SDL_RenderPresent(renderer);

  // TODO: cleanup
}
```

When linking with Emscripten, you need to use `-s USE_SDL=2`. This will tell Emscripten to fetch the SDL2 library, already precompiled to WebAssembly, and link it with your main application.

```
emcc example.cpp -o example.html -s USE_SDL=2
```

When the example is loaded in the browser, you'll see the familiar green rectangle:

Emscripten-generated HTML page showing a green rectangle on a black square canvas.

This code has a couple of problems though. First, it lacks proper cleanup of allocated resources. Second, on the web, pages don't get closed automatically when an application has finished its

execution, so the image on the canvas gets preserved. However, when the same code is recompiled natively with

```
clang example.cpp -o example -lSDL2
```

and executed, the created window will only blink briefly and immediately close upon exit, so the user doesn't have a chance to see the image.

Integrating an event loop

A more complete and idiomatic example would look need to wait in an event loop until the user chooses to quit the application:

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

int main() {
  SDL_Init(SDL_INIT_VIDEO);

  SDL_Window *window;
  SDL_Renderer *renderer;
  SDL_CreateWindowAndRenderer(300, 300, 0, &window, &renderer);

  SDL_SetRenderDrawColor(renderer, /* RGBA: green */ 0x00, 0x80, 0x00, 0xFF);
  SDL_Rect rect = {.x = 10, .y = 10, .w = 150, .h = 100};
  SDL_RenderFillRect(renderer, &rect);
  SDL_RenderPresent(renderer);

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

  SDL_DestroyRenderer(renderer);
  SDL_DestroyWindow(window);

  SDL_Quit();
}
```

After the image has been drawn to a window, the application now waits in a loop, where it can process keyboard, mouse and other user events. When the user closes the window, they'll trigger an `SDL_QUIT` event, which will be intercepted to exit the loop. After the loop is exited, the application will do the cleanup and then exit itself.

Now compiling this example on Linux works as expected and shows a 300 by 300 window with a green rectangle:

A square Linux window with black background and a green rectangle.

However, the example no longer works on the web. The Emscripten-generated page freezes immediately during the load and never shows the rendered image:

Emscripten-generated HTML page overlaid with a 'Page Unresponsive' error dialogue suggesting to either wait for the page to become responsible or exit the page

What happened? I'll quote the answer from the article "Using asynchronous web APIs from WebAssembly":

> The short version is that the browser runs all the pieces of code in sort of an infinite loop, by taking them from the queue one by one. When some event is triggered, the browser queues the corresponding handler, and on the next loop iteration it's taken out from the queue and executed. This mechanism allows simulating concurrency and running lots of parallel operations while using only a single thread.
>
> The important thing to remember about this mechanism is that, while your custom JavaScript (or WebAssembly) code executes, the event loop is blocked [...]

The preceding example executes an infinite event loop, while the code itself runs inside another infinite event loop, implicitly provided by the browser. The inner loop never relinquishes control to the outer one, so the browser doesn't get a chance to process external events or draw things onto the page.

There are two ways to fix this problem.

Unblocking event loop with Asyncify

First, as described in the linked article, you can use Asyncify. It's an Emscripten feature that allows to "pause" the C or C++ program, give control back to the event loop, and wake up the program when some asynchronous operation has finished.

Such asynchronous operation can be even "sleep for the minimum possible time", expressed via `emscripten_sleep(0)` API. By embedding it in the middle of the loop, I can ensure that the control is returned to browser's event loop on each iteration, and the page remains responsive and can handle any events:

```
#include <SDL2/SDL.h>
#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#endif

int main() {
  SDL_Init(SDL_INIT_VIDEO);

  SDL_Window *window;
  SDL_Renderer *renderer;
  SDL_CreateWindowAndRenderer(300, 300, 0, &window, &renderer);

  SDL_SetRenderDrawColor(renderer, /* RGBA: green */ 0x00, 0x80, 0x00, 0xFF);
  SDL_Rect rect = {.x = 10, .y = 10, .w = 150, .h = 100};
  SDL_RenderFillRect(renderer, &rect);
  SDL_RenderPresent(renderer);

  while (1) {
    SDL_Event event;
    SDL_PollEvent(&event);
    if (event.type == SDL_QUIT) {
      break;
    }
#ifdef __EMSCRIPTEN__
    emscripten_sleep(0);
#endif
  }

  SDL_DestroyRenderer(renderer);
  SDL_DestroyWindow(window);

  SDL_Quit();
}
```

This code now needs to be compiled with Asyncify enabled:

```
emcc example.cpp -o example.html -s USE_SDL=2 -s ASYNCIFY
```

And the application works as expected on the web again:

Emscripten-generated HTML page showing a green rectangle on a black square canvas.

However, Asyncify can have non-trivial code size overhead. If it's only used for a top-level event loop in the application, a better option can be to use the `emscripten_set_main_loop` function.

Unblocking event loop with "main loop" APIs

`emscripten_set_main_loop` doesn't require any compiler transformations for unwinding and rewinding the call stack, and that way avoids the code size overhead. However, in exchange, it requires a lot more manual modifications to the code.

First, the body of the event loop needs to be extracted into a separate function. Then, `emscripten_set_main_loop` needs to be called with that function as a callback in the first argument, an FPS in the second argument (`0` for the native refresh interval), and a boolean indicating whether to simulate infinite loop (`true`) in the third:

```
emscripten_set_main_loop(callback, 0, true);
```

The newly created callback won't have any access to the stack variables in the `main` function, so variables like `window` and `renderer` need to be either extracted into a heap-allocated struct and its pointer passed via `emscripten_set_main_loop_arg` variant of the API, or extracted into global `static` variables (I went with the latter for simplicity). The result is slightly harder to follow, but it draws the same rectangle as the last example:

```
#include <SDL2/SDL.h>
#include <stdio.h>
#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#endif

SDL_Window *window;
SDL_Renderer *renderer;

bool handle_events() {
  SDL_Event event;
  SDL_PollEvent(&event);
  if (event.type == SDL_QUIT) {
    return false;
  }
```

```
  return true;
}

void run_main_loop() {
#ifdef __EMSCRIPTEN__
  emscripten_set_main_loop([]() { handle_events(); }, 0, true);
#else
  while (handle_events())
    ;
#endif
}

int main() {
  SDL_Init(SDL_INIT_VIDEO);

  SDL_CreateWindowAndRenderer(300, 300, 0, &window, &renderer);

  SDL_SetRenderDrawColor(renderer, /* RGBA: green */ 0x00, 0x80, 0x00, 0xFF);
  SDL_Rect rect = {.x = 10, .y = 10, .w = 150, .h = 100};
  SDL_RenderFillRect(renderer, &rect);
  SDL_RenderPresent(renderer);

  run_main_loop();

  SDL_DestroyRenderer(renderer);
  SDL_DestroyWindow(window);

  SDL_Quit();
}
```

Since all the control flow changes are manual and reflected in the source code, it can be compiled without the Asyncify feature again:

```
emcc example.cpp -o example.html -s USE_SDL=2
```

This example might seem useless, because it works no differently from the first version, where the rectangle was drawn on canvas successfully despite the code being a lot simpler, and the `SDL_QUIT` event—the only one handled in the `handle_events` function—is ignored on the web anyway.

However, proper event loop integration - either via Asyncify or via `emscripten_set_main_loop` - pays off if you decide to add any kind of animation or interactivity.

Handling user interactions

For example, with a few changes to the last example you can make the rectangle move in response to keyboard events:

```
#include <SDL2/SDL.h>
#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#endif

SDL_Window *window;
SDL_Renderer *renderer;

SDL_Rect rect = {.x = 10, .y = 10, .w = 150, .h = 100};

void redraw() {
  SDL_SetRenderDrawColor(renderer, /* RGBA: black */ 0x00, 0x00, 0x00, 0xFF);
  SDL_RenderClear(renderer);
  SDL_SetRenderDrawColor(renderer, /* RGBA: green */ 0x00, 0x80, 0x00, 0xFF);
  SDL_RenderFillRect(renderer, &rect);
  SDL_RenderPresent(renderer);
}

uint32_t ticksForNextKeyDown = 0;

bool handle_events() {
  SDL_Event event;
  SDL_PollEvent(&event);
  if (event.type == SDL_QUIT) {
    return false;
  }
  if (event.type == SDL_KEYDOWN) {
    uint32_t ticksNow = SDL_GetTicks();
    if (SDL_TICKS_PASSED(ticksNow, ticksForNextKeyDown)) {
      // Throttle keydown events for 10ms.
      ticksForNextKeyDown = ticksNow + 10;
      switch (event.key.keysym.sym) {
        case SDLK_UP:
          rect.y -= 1;
          break;
```

```
        case SDLK_DOWN:
          rect.y += 1;
          break;
        case SDLK_RIGHT:
          rect.x += 1;
          break;
        case SDLK_LEFT:
          rect.x -= 1;
          break;
      }
      redraw();
    }
  }
  return true;
}

void run_main_loop() {
#ifdef __EMSCRIPTEN__
  emscripten_set_main_loop([]() { handle_events(); }, 0, true);
#else
  while (handle_events())
    ;
#endif
}

int main() {
  SDL_Init(SDL_INIT_VIDEO);

  SDL_CreateWindowAndRenderer(300, 300, 0, &window, &renderer);

  redraw();
  run_main_loop();

  SDL_DestroyRenderer(renderer);
  SDL_DestroyWindow(window);

  SDL_Quit();
}
```

Drawing other shapes with SDL2_gfx

SDL2 abstracts away cross-platform differences and various types of media devices in a single API, but it's still a pretty low-level library. In particular for graphics, while it provides APIs for drawing points, lines and rectangles, implementation of any more complex shapes and transformations is left to the user.

SDL2_gfx is a separate library that fills that gap. For example, it can be used to replace a rectangle in the example above with a circle:

```
#include <SDL2/SDL.h>
#include <SDL2/SDL2_gfxPrimitives.h>
#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#endif

SDL_Window *window;
SDL_Renderer *renderer;

SDL_Point center = {.x = 100, .y = 100};
const int radius = 100;

void redraw() {
  SDL_SetRenderDrawColor(renderer, /* RGBA: black */ 0x00, 0x00, 0x00, 0xFF);
  SDL_RenderClear(renderer);
  filledCircleRGBA(renderer, center.x, center.y, radius,
                   /* RGBA: green */ 0x00, 0x80, 0x00, 0xFF);
  SDL_RenderPresent(renderer);
}

uint32_t ticksForNextKeyDown = 0;

bool handle_events() {
  SDL_Event event;
  SDL_PollEvent(&event);
  if (event.type == SDL_QUIT) {
    return false;
  }
  if (event.type == SDL_KEYDOWN) {
    uint32_t ticksNow = SDL_GetTicks();
```

```
    if (SDL_TICKS_PASSED(ticksNow, ticksForNextKeyDown)) {
      // Throttle keydown events for 10ms.
      ticksForNextKeyDown = ticksNow + 10;
      switch (event.key.keysym.sym) {
        case SDLK_UP:
          center.y -= 1;
          break;
        case SDLK_DOWN:
          center.y += 1;
          break;
        case SDLK_RIGHT:
          center.x += 1;
          break;
        case SDLK_LEFT:
          center.x -= 1;
          break;
      }
      redraw();
    }
  }
  return true;
}

void run_main_loop() {
#ifdef __EMSCRIPTEN__
  emscripten_set_main_loop([]() { handle_events(); }, 0, true);
#else
  while (handle_events())
    ;
#endif
}

int main() {
  SDL_Init(SDL_INIT_VIDEO);

  SDL_CreateWindowAndRenderer(300, 300, 0, &window, &renderer);

  redraw();
  run_main_loop();
```

```
  SDL_DestroyRenderer(renderer);
  SDL_DestroyWindow(window);

  SDL_Quit();
}
```

Now the SDL2_gfx library also needs to be linked into the application. It's done similarly to SDL2:

```
# Native version
$ clang example.cpp -o example -lSDL2 -lSDL2_gfx
# Web version
$ emcc --bind foo.cpp -o foo.html -s USE_SDL=2 -s USE_SDL_GFX=2
```

And here are the results running on Linux:

A square Linux window with black background and a green circle.

And on the web:

Emscripten-generated HTML page showing a green circle on a black square canvas.

For more graphics primitives, check out the auto-generated docs.

□□□ #3
□ □□□ □□□ 2026-10-31 21:50:41 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:04 UTC