Porting an existing Game via Emscripten

After investigating the inner workings of LLVM and Emscripten we will now attempt to port an existing program using the Emscripten toolchain. The program which will be used is a small game written by the author using SDL2[https://www.libsdl.org/index.php] and SDL2\_GFX[http://www.ferzkopp.net/wordpress/2016/01/02/sdl\_gfx-sdl2\_gfx/]. The source code from the game can be obtained from <https://github.com/in0x/thesis-emscripten-game>.

// Place screenshot of game running here //

Figure: A Screenshot of the native binary of the game running

First we will try to to invoke em++ to compile our codebase. Since em++’s options are very similar to g++, the two compilers can often succesfully be invoked with the same command.

g++ \*.cpp -std=c++11 -I/usr/local/include/SDL2 -lSDL2\_gfx -lSDL2 -o game

// The command to compile with g++, taken from the projects makefile –listing 1

em++ -s USE\_SDL=2 -I/usr/local/include/SDL2 -std=c++11 C++/\*.cpp -o game.html -O3

// The command to compile via Emscripten to a HTML target –listing 2

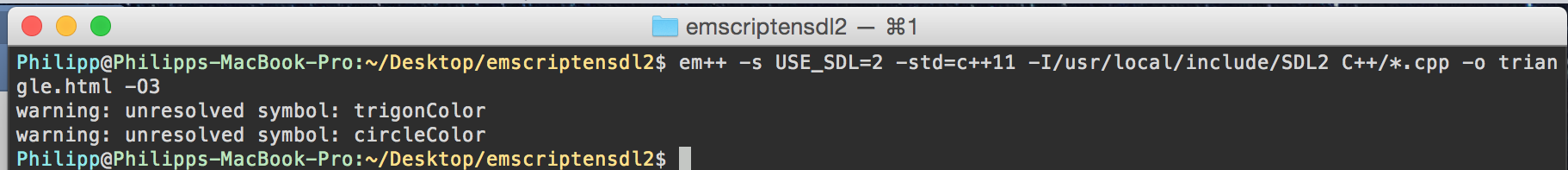
Note that the flag -s USE\_SDL=2 is used which allows Emscripten to be told that SDL2 should be used when building this projects as opposed to SDL1, which ships with Emscripten per default. [https://kripken.github.io/emscripten-site/docs/compiling/Building-Projects.html#emscripten-ports]

This flag triggers a download from the Emscripten Ports repo, which is a collection of libraries ported to Emscripten, that are integrated with emcc. These also include zlib, the Vorbis audio codec and the Bullet physics engine, among others.[https://github.com/emscripten-ports]

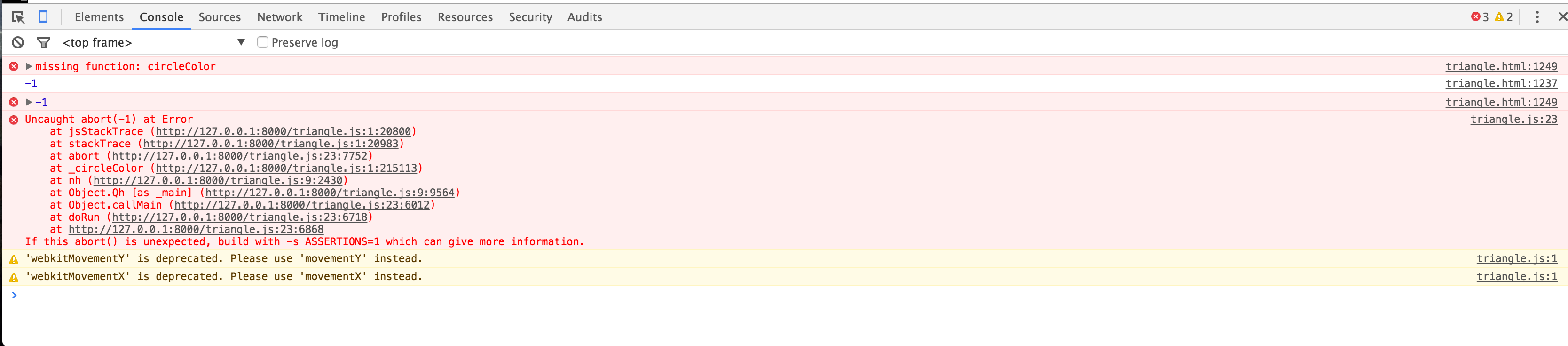
The linker commands –l can be omitted, as Emscripten wont be using the precompiled C++ libraries.

We also use the flag –O3, which tells emcc to apply the most aggressive level of optimization. [https://kripken.github.io/emscripten-site/docs/tools\_reference/emcc.html]

Running this command results in the output shown in Figure 2.



TrigonColor and circleColor are both functions included from SDL2\_GFX. This is problematic because SDL2\_GFX does not ship with SDL2 by default, but rather is a library provided by a third party. Since Emscripten only has access to vanilla SDL2, SDL2\_GFX will need to be provided externally.



// Figure 3: Inspecting the Google Chrome Developer Console provides more information about how and when the build failed. //

In the simplest case, the library’s source code is freely available, so that it can be built together with the project in question. Additionally, it would also be desirable if the amount of code to be compiled is small, so that long compilation times can be prevented. Luckily both these are the case with SDL2\_GFX. The libraries source files can be copied into the project and em++ can also be told to look for the files. This resolves the linker’s unresolved symbol errors.

If it would be unreasonable to occasionally have to recompile the projects supporting libraries, Emscripten also supports generating and linking objects files.

[https://github.com/kripken/emscripten/wiki/Linking]. One can build both LLVM bitcode files or complete JavaScript modules and then link these into the main module (i.e. the main programming which is supported by the libraries) while building the project.

When invoked with a –o argument without a file extension, emcc produces a binary file.

Invoking the command in listing 3 produces such a binary file of SDL2\_GFX. This file actually contains the bytecode representation of the clang-compiled LLVM library. Adding the extension .bc to the argument results in the same file being produced and is preferable, as emcc will not use a file without an extension.

emcc –O2 -s USE\_SDL=2 \*.c -o sdl2\_gfx –listing 3, the command to generate an LLVM bytecode file of SDL2\_GFX

Adding the name of this file into our compiler options will now link the precompiled build of SDL2\_GFX into the game’s build.

Having now compiled SDL2\_GFX into our Emscripten build, the compilation succeeds and results in an HTML file that executes the build.

// Insert image of failed build running here //

However, once opened, the page immediately freezes and goes into a dead-lock. The explanation for this can be found by investigating deeper into how native C++ applications usually handle continuously looped programs compared to a browser’s JavaScript engine.

// Subchapter Processes on Native OSs versus the Browser

Consider the following snippet of code, shown in listing 3

**void** render**()**

**{**

**while** **(**true**)**

**{**

*/\* Do rendering \*/*

**}**

**}**

**int** main**()**

**{**

render**();**

**return** **0;**

**}**

Listing 3: A continuous loop in C

Assuming the application built from this code were to have exclusive access to the system once started, it would cause all other activities running on the system to freeze, until the program has finished running. The application would for example not be able to poll the system’s event loop for HID input.

This is not the case on modern operating systems. If started by the user, the process will not cause all other activities on the system to pause. Instead, the user is still capable of minimizing or ending the task and using other programs.

This is possible because of the work done by modern operating systems for the user. OSs use an Abstraction called a Process to manage programs.

It allows the OS to encapsulate the information associated with the process, such as the start of its memory block or which IO sources the process it is reading from, as well as its state, i.e. running, blocked (reading from an IO source) or waiting for execution (Arpaci-Dusseau and Arpaci-Dusseau 2015, 26). Using this abstraction the OS (Footnote:The OS consists of many complex parts that each have their own part in managing the system. Investigating these in detail would be beyond the scope of this chapter, however.) can allocate time for when each process runs and waits as well as killing of frozen processes.