Ported To
VimConf 2018 at Tokyo
• Loves programming tools such as programming languages or editors

• 70+ Vim plugins
  • clever-f.vim,vim-clang-format, committia.vim, vim-grammarous etc)
  • Maintainer of filetype=wast

• Editor frontend such as NyaoVim, vim.wasm (Maintainer of Neovim Node.js binding)

• Creates my own language with LLVM
Today I'll talk about compiling Vim source codes into WebAssembly. Vim is working on browsers.
ONLINE DEMO

https://rhysd.github.io/vim.wasm/
Agenda

• What's WebAssembly?
• Details of Implementation
• What is hard?
• Impressions and Future Works
What's WebAssembly (Wasm)?
WebAssembly (Wasm)

• New programming language running on browsers (on Stack-based VM)

• Programs are in binary format supposed to be compiled from other languages such as C, C++, Rust into Wasm

• Wasm language spec is defined by W3C as long-live web standard

• Comparing to JavaScript, Wasm is faster, memory efficient, file size efficient and safer

• Chrome, Firefox, Safari, Edge already support Wasm

• https://webassembly.org/
• https://developer.mozilla.org/docs/WebAssembly
• https://webassembly.github.io/spec/core/index.html#
WebAssembly (Wasm)

- Example: C source into Wasm (text format)

```c
#include <stdio.h>

int main(int argc, char ** argv) {
    puts("hello world\n");
}
```

```wasm
(module
    ; ; ...
    (data (i32.const a) "hello world")
    ; ; ...
    (func $_main (; 18 ;) (param $0 i32) (param $1 i32) (result i32)
        (drop
            (call $_puts
                (i32.const 1152)
            )
        )
        (i32.const 0)
    )
)
```

$ emcc -O3 -g hello.c

↑ compiling with emscripten

a.out.wast
Wasm is supposed to be compiled from other languages, compiler toolchain is necessary

emscripten: A toolchain to compile C, C++ sources into Wasm

Compiler&Linker: Build multiple C, C++ files into one Wasm file

Runtime: There are no malloc, IO, syscalls on browsers. Runtime libraries to shim them

Support interoperability between C, C++ and JavaScript. It enables to call functions from each other
# emcc: C compiler. Usage is similar to gcc
$ emcc -O3 hello.c -o hello.html

# Followings are generated
# - hello.html (entry point. Open in browser)
# - hello.js (JavaScript runtime)
# - hello.wasm (Compilerd Wasm from hello.c containing _main() function)

$ python3 -m http.server 1234
$ open localhost:1234/hello.html
vim.wasm

Details of Implementation
What's vim.wasm?

A fork of Vim. Using emscripten, Vim's C sources are compiled into Wasm. It allows Vim to run on browsers.

Only supports tiny features yet.

- Repository: https://github.com/rhysd/vim.wasm
- Japanese blog: https://rhysd.hatenablog.com/
What's vim.wasm?

All source code changes:

https://goo.gl/5WMW9n
Policy of implementation

• emscripten provides Unix-like runtime environment. But many things are missing on environment where Wasm is running.
  • Stdin is missing (stdout is connected to console.log)
  • Terminal screen is missing
  • Terminal library such as curses is missing
  • Wasm can't call DOM APIs (can't access to DOM elements)
Policy of implementation

• Implement Wasm Vim frontend as one of GUI frontends. Never run CUI Vim

• Use Core parts of Vim as-is (rendering process, input buffering, etc etc...)

• Create JavaScript runtime for doing what Wasm can't do. It collaborates with C functions

• To follow upstream changes, basically avoid breaking code changes by switching implementation with C preprocessor
Build: Fix autoconf settings

- Add variables such as $WASM_SRC to src/ Makefile
- Add gui_wasm.c and other sources for Wasm to dependencies

```
WASM_SRC = gui.c gui_wasm.c
WASM_OBJ = objects/gui.o objects/gui_wasm.o
WASM_DEFS = -DFEAT_GUI_WASM
WASM_IPATH = -I. -Iproto
WASM_LIBS_DIRS =
WASM_LIBS1 =
WASM_LIBS2 =
WASM_INSTALL = install_wasm
WASM_TARGETS =
WASM_MAN_TARGETS =
WASM_TESTTARGET =
WASM_BUNDLE =
WASM_TESTARG =

# ...

objects/gui_wasm.o: gui_wasm.c
  $(CCC) -o $@ gui_wasm.c
```
Build: Fix autoconf settings

• emconfigure, a wrapper script of ./configure, configures everything for emcc (C compiler of emscripten)

• To build minimal Vim, specify all `--disable-*` and `--with-features=tiny` on running ./configure

• ./configure fails at first. Repeat fixing src/auto/configure directly and trying again
  
  • e.g. Ignore terminal library check. It is mandatory for normal Vim but we don't need

• After configure passes, try `make` → It builds src/vim.bc

• https://kripken.github.io/emscripten-site/docs/compiling/Building-Projects.html
Overview of implementation

JavaScript
- Calculate keycode of input char
- Render to <canvas/>

GUI Implementation in C
- Call C functions
- Calculate how to render canvas
- Call rendering functions
- Add key sequence to input buffer

Canvas API
- Call JS functions

Keyboard event listener
- vim.html
  - KeyboardEvent
  - <canvas/>

Input:
Output:

add_to_input_buf()
Overview of implementation

wasm/runtime.js

src/gui_wasm.c

wasm/vim.html

Vim Core

KeyboardEvent
Overview of implementation

wasm/runtime.js
Calculate keycode of input char
Keyboard event listener

wasm/vim.html

src/gui_wasm.c

Vim Core
Overview of implementation

- **wasm/runtime.js**
  - Calculate keycode of input char

- **gui_wasm_send_key()**

- **wasm/vim.html**
  - Keyboard event listener
  - `KeyboardEvent`

- **src/gui_wasm.c**
  - Calculate key sequence

- **Vim Core**
Overview of implementation

wasm/runtime.js
Calculate keycode of input char

gui_wasm_send_key()

src/gui_wasm.c
Calculate key sequence

Vim Core
Add key sequence to input buffer

wasm/vim.html
Keyboard event listener

KeybaordEvent

Input:
Overview of implementation

Output:

- `wasm/runtime.js`
- `src/gui_wasm.c`
- `wasm/vim.html`
- `Vim Core`

Rendering events to screen happen
Overview of implementation

wasm/runtime.js

src/gui_wasm.c

Calculate how to render canvas

gui_mch_*(), ...

Vim Core

wasm/vim.html

Rendering events to screen happen
Overview of implementation

wasm/runtime.js

Render to <canvas/>
vimwasm_(*)

src/gui_wasm.c

Calculate how to render canvas
gui_mch_(*), ...

wasm/vim.html

Vim Core

Rendering events to screen happen

Output:
Overview of implementation

wasm/runtime.js

Render to <canvas/>

vimwasm_*( )

Canvas API

wasm/vim.html

src/gui_wasm.c

Calculate how to render canvas
gui_mch_*( ), ...

Vim Core

Rendering events to screen happen

Output:
Overview of implementation

JavaScript
- Calculate keycode of input char
- Render to <canvas/>

Canvas API
- Call JS functions
- Call rendering functions

vim.html
- Keyboard event listener

GUI Implementation in C
- Calculate key sequence
- Calculate how to render canvas

Vim Core
- Rendering events to screen happen
- Add key sequence to input buffer

Input: →
Output: ←

Call C functions
add_to_input_buf()
GUI implementation of Vim

• Vim supports GTK, GNOME, MSWIN... Only one of them can be enabled at once. It can be determined at running .configure

• GUI frontends are implemented in src/gui_*.{c,h} (and other files with C preprocessor)

• e.g. `--enable-gui=gtk3` compiles src/gui_gtk.c and gui_gtk.o will be linked (Other GUI implementations are ignored)
### src/gui_*.c implementations

- Vim properly calls specific functions defined in gui_*.c
- gui_*.c implements the functions with GUI libraries/frameworks (rendering screen, wait for input from user, …)
- Example of functions:

<table>
<thead>
<tr>
<th>Function name</th>
<th>What should be done</th>
</tr>
</thead>
<tbody>
<tr>
<td>gui_mch_init</td>
<td>Set default highlight colors, window size, …</td>
</tr>
<tr>
<td>gui_mch_set_fg_color</td>
<td>Set current foreground/background colors</td>
</tr>
<tr>
<td>gui_mch_set_bg_color</td>
<td></td>
</tr>
<tr>
<td>gui_mch_draw_string</td>
<td>Render a text at (row, col) with current foreground color</td>
</tr>
<tr>
<td>gui_mch_clear_block</td>
<td>Clear a rectangle (row1, col1, row2, col2) by painting the rectangle with tcurrent background color</td>
</tr>
<tr>
<td>gui_mch_delete_lines</td>
<td>Delete specified number of lines at specified row (and scroll up lines)</td>
</tr>
<tr>
<td>gui_wasm_resize_shell</td>
<td>Update screen size with specified rows and columns</td>
</tr>
<tr>
<td>gui_mch_wait_for_chars</td>
<td>Wait user’s input by polling and blocking</td>
</tr>
</tbody>
</table>
The information to render things is passed via function parameters. Rendering functions in gui_wasm.c calculates how they are rendered on <canvas/>. And pass the result to JavaScript functions.
Example: Clear rect (gui_mch_clear_block)

```c
void gui_mch_clear_block(int row1, int col1, int row2, int col2)
{
    // Set default background color (gui.bg_color_code will also be set)
    gui_mch_set_bg_color(gui.back_pixel);

    // Vim handles rendering information with (row,col), but <canvas/> handles coordinates (x,y). Translate (row,col) into (x,y)
    int x = gui.char_width * col1;
    int y = gui.char_height * row1;
    int w = gui.char_width * (col2 - col1 + 1);
    int h = gui.char_height * (row2 - row1 + 1);

    // <canvas/> handles colors with color code such as #123456
    char *color_code = gui.bg_color_code;
    int filled = TRUE;

    // Clear a block by painting a rectangle with background color
    // vimwasm_* functions are declared, but not defined in C
    // Thanks to emscripten, it calls a function in JavaScript
    vimwasm_draw_rect(x, y, w, h, color_code, filled);
}
```
Example: Render a text (gui_mch_draw_string)

```c
// s: text to render, len: length of text, flags: attributes of rendering
void
gui_mch_draw_string(int row, int col, char_u *s, int len, int flags)
{
    // Clear text region by background color if not transparent
    if (!(flags&DRAW_TRANSP)) {
        draw_rect(row, col, row, col + len - 1, gui.bg_color_code, TRUE);
    }

    // If the text only contains white spaces, don't need to render it
    // In the case return early...

    // Call JavaScript side function. Pass all information required to render the text
    vimwasm_draw_text(
        gui.font_height,       // line height
        gui.char_height,      // character height
        gui.char_width,       // character width
        gui.char_width * col, // x
        gui.char_height * row, // y
        (char *)s,            // text
        len,                  // length of text
        flags&DRAW_BOLD,      // bold or not
        flags&DRAW_UNDERL,    // underline or not
        flags&DRAW_UNDERC,    // undercurl or not
        flags&DRAW_STRIKE);  // strikethrough or not
}
```
Key input is received by a browser. JavaScript side sends the key input to C function (in Wasm). In C, calculate key input sequence and add it to Vim's input buffer.
Example: Hanle key input (gui_wasm_send_key)

```c
// Function called by JavaScript on key input
// - key_code: key code calculated in JavaScript
// - special_code: Special code for special character such as arrow keys
// - ctrl_key: Ctrl key is pressed or not
// - shift_key: Shift key is pressed or not
// - alt_key: Alt key is pressed or not
// - meta_key: Meta (Cmd) key is pressed or not

void gui_wasm_send_key(int key_code, int special_code, int ctrl_key, int shift_key, int alt_key, int meta_key)
{
    // Create a modifier keys mask with MOD_MASK_CTRL, MOD_MASK_SHIFT, ...
    // If special_code is non-zero, encode special code into key_code with TO_SPECIAL() macro...
    // If <C-c>, set interrupt flag...
    // Apply the modifier keys mask to key_code by extract_modifiers()...

    short len = 0;  // Length of sequence
    char_u input[20];  // Actual input sequence

    // If any modifier key is pressed, add modifier key sequence at first...
    if (IS_SPECIAL(key_code)) {
        // Add key sequence for special codes...
    } else {
        input[len++] = key_code;  // Add normal key to input sequence
    }

    // Add the input sequence into Vim's input buffer
    // Vim will pick up the inputs from the buffer and process them
    add_to_input_buf(input, len);
}
```
Overview of implementation

- **JavaScript**
  - Calculate keycode of input char
  - Render to `<canvas/>`
  - Call JS functions

- **GUI Implementation in C**
  - Calculate key sequence
  - Calculate how to render canvas
  - Call rendering functions
  - Call C functions
  - add_to_input_buf()

- **Canvas API**

- **Vim Core**
  - Rendering events to screen happen
  - Add key sequence to input buffer

- **Input**
- **Output**

- **Keyboard event listener**
- **vim.html**
  - KeyEvent
  - `<canvas/>`
JavaScript Runtime

• Define JavaScript functions to be called from C or to handle key input events in `wasm/runtime.js`

• Actual implementation is in TypeScript for maintainability

• emscripten provides the way to create JavaScript library called from C

$ emcc --pre-js pre.js --js-library runtime.js
const VimWasmRuntime = {
  $VW__postset: 'VW.init()',
  $VW: {
    init() {
      class VimWindow {
        /* Window size management */
      }

      class VimInput {
        onVimInit() {
          const paramTypes = ['number', 'number', 'number', 'number', 'number', 'number'];
          VimInput.prototype.sendKeyToVim = Module.cwrap('gui_wasm_send_key', null, paramTypes);
        }

        onKeydown(event) {
          // Function called on keydown event.
          // This calculates key code from KeyboardEvent and send it to C via VimInput.prototype.sendKeyToVim
        }
      }

      class CanvasRenderer {
        /* Class for <canvas/> rendering. Enque rendering events and handle them on animation frame */
      },
    },
  },

  // Define functions called from C

  // void vimwasm_draw_text(int, int, int, int, int, char *, int, int, int, int, int);
  vimwasm_draw_text(charHeight, lineHeight, charWidth, x, y, str, len, bold, underline, undercurl, strike) {
    const text = Pointer_stringify(str, len); // Convert C pointer into JavaScript string
    VW.renderer.enqueue(VW.renderer.drawText, /*...*/); // Enque the rendering event into render queue
  },

  // Other functions called from C...
};
autoAddDeps(VimWasmRuntime, '$VW');
mergeInto(LibraryManager.library, VimWasmRuntime); // Register as JavaScript library via emscripten API
Overview of implementation

JavaScript
- Calculate keycode of input char
- Render to `<canvas/>`
- Call JS functions

GUI Implementation in C
- Calculate key sequence
- Call rendering functions
- Add key sequence to input buffer

Canvas API
- Call C functions

Keyboard event listener

vim.html
- `<canvas/>`
- `KeyboardEvent`
vim.html

• Entrypoint of this application. Users open with browser

• emcc will generate an HTML file from HTML template

• Give an HTML template file path to --shell-file option of emcc

https://kripken.github.io/emscripten-site/docs/tools_reference/emcc.html
Files in runtime/*

• To reduce total file size, only minimal colorscheme, syntax files and vimrc are included

• emcc provides --preload-file option. It bundles all static files as one binary file

• Vim can access to files in /usr/local/share/vim via emscripten's FileSystem API as normal files. So no modification is needed for file access.

Overview of implementation

### wasm/runtime.js
- **Calculate keycode of input char**
- **Render to `<canvas/>`**

### src/gui_wasm.c
- **Calculate key sequence**
- **Calculate how to render canvas**
- **gui_mch_*( ), ...**
- **add_to_input_buf()**

### wasm/vim.html
- **Keyboard event listener**
- **Canvas API**
- **KeybaordEvent**
- `<canvas/>`
main_loop()  normal.c  ui.c  getchar.c  gui_wasm.c  runtime.js  Browser  You

normal_cmd()  ui_inchar()  safe_vgetc()  gui_mch_wait_for_chars()  Input polling

Add key seq to input buffer  add_to_input_buf()  gui_wasm_send_key()  Calculate key sequence

VimInput.onVimInput()  Dispatch KeyboardEvent

input key  Calculate key code
main_loop() -> normal.cmd() -> ui.inchar() -> safe_vgetc() -> gui.mch.wait_for_chars() -> gui.wasm.send_key() -> Input polling -> Calculate key code

Add key seq to input buffer

vim.is_input_buf_empty() -> input_available() -> gui.mch.wait_for_chars() -> gui.wasm.send_key() -> Input polling -> Calculate key sequence

Process input by Vim core

Call JS functions e.g. vim.wasm.draw_text()

Call rendering functions e.g. gui.mch.draw_string()

> On animation frame

Enqueue event to rendering queue

Render to <canvas/> -> Display
vim.wasm
what is hard
Debugging is hard

- Browser’s DevTools don’t fully support Wasm code debugging (e.g., breaking points)

  - Stepping over debugging (Chrome 70) and sourcemap (Chrome 71) will be supported! 🎉

- Only way to debug was printing. I added many logs to analyze bugs

  - In C: Output logs on debug build by switching by C preprocessor

  - In JavaScript: By switching an HTML files on debug/release build, they switch to enable/disable debug log

- `<canvas/>` is hard to debug since we can only see the rendered result image
Wasm can't sleep()

- Wasm is designed not to do blocking things the same as JavaScript. (If it's possible, it means blocking main thread)

- But the Vim main loop requires blocking wait to wait for user input in gui_*.c. (gui_mch_wait_for_chars() is expected to do input polling with input_available(). Polling requires sleep() to reduce CPU usage.)
Somehow can we do sleep()?

- **Idea1: Busy loop**  Busy loop in Wasm prevents keyboard event listeners being called. Key input does not work

- **Idea2: Calls sync XHR and wait cache response in ServiceWorker**  Chrome does not support this due to bug. Firefox works but it causes high CPU usage of browser process.
Somehow can we do sleep()?

- emscripten's `emscripten_sleep()`

- By adding `-s EMTERPRETIFY=1` to `emcc`, special function `emscripten_sleep()` will be available

- It works like normal sleep() function 😃😭😢
What's emscripten_sleep()

• But Wasm cannot block by design. How blocking function emscripten_sleep() is implemented? → Actually it does not block, but it appears to block

• Functions including emscripten_sleep() and functions calling them are compiled to Emterpreter byte codes, not wasm directly. The byte codes are executed by an interpreter called Emterpreter. (Other functions are compiled to Wasm as usual)

• Emterpreter is interpreter. So it can suspend execution and resume the execution after. Emterpreter suspends the execution of function at the call of emscripten_sleep(), stores the execution state, wait for duration asynchronously, and resumes the execution state to continue to run

https://github.com/kripken/emscripten/wiki/Emterpreter
emscripten_sleep()

• Explaining what happens by pseudo code (JavaScript).

C source

```c
printf("wait for 100ms\n");
int i = 42;
emscripten_sleep(100);
int j = i + 10;
printf("result=%d\n", j);
```

emcc transforms code on compilation

```
$ emcc -s EMTERPRETIFY=1
```

```javascript
// Actually empreterer is run in Wasm and wait asynchronously in JavaScript. For explanation I wrote below code in JavaScript

empreterer = new Empreterer();

// Run codes before emscripten_sleep()
// on interpreter
empreterer.run_code(`
    printf("wait for 100ms\n");
    int i = 42;
`);

// Suspend execution of interpreter
const state = empreterer.suspend();

// Wait 100ms asynchronously with timer
setTimeout(function() {
    // Resume suspended execution state
    empreterer.resume(state);

    // Run codes after emscripten_sleep()
    // on interpreter
    empreterer.run_code(`
        int j = i + 10;
        printf("result=%d\n", j);
    `);
}, 100);
```
Problem: Emterpreter is Unsatable

- It makes compilation with emcc much slower for heavy code transformation
- Functions should be run with Emterpreter must be specified manually as option of emcc
- Emterpreter byte code is very redundant. It makes binary size bigger
- Passing strings from JavaScript functions to C functions which are run on Emterpreter causes crash. JS functions can't pass strings to C functions
vim.wasm
Impressions and
Future Works
Impressions

- emscripten and Wasm implementation of browsers works fine 🎉
- 'tiny' feature is so helpful as start point of porting
- Without modifying core of Vim editor, only less than 7000 lines modifications enabled this porting. It only took 8 days to see Vim works on browser at first!
- I learnt a log about Wasm toolchain and Vim's GUI implementation by this project
Future Works

- Implement Vim main loop without Emterpreter (trying in async-eventloop branch)
  - Vim main loop needs to be asynchronous (using emscripten main loop support)
  - It's hard to rewrite C sync function with async function with callbacks hell
    - e.g. `int foo(char*)` → `void foo_async(char*, void (*)(int))`
- Support small feature for syntax highlight and support mouse and IME
- Distribute `vim.wasm` as Web Component. People would be eble to use vim.wasm easily in browsers

https://github.com/rhysd/vim.wasm/tree/async-eventloop
https://kripken.github.io/emscripten-site/docs/porting/emscripten-runtime-environment.html#browser-main-loop