
# Bringing Your C/C++ Game to the Web via Emscripten

Vladimir Vukićević
**mozilla**

Please **Tweet** This.
And **Facebook**.
And **Tumblr**.
And **Blog**.

# On Deck

Brief Overview of
Emscripten & asm.js

Compiling to the Web:
Getting Started is Easy!

How Do I Deploy It?

mozilla

# Emscripten & asm.js

A Brief Overview

# Emscripten

Emscripten is a tool for
compiling C/C++ to JavaScript

Built on top of LLVM/Clang

All LLVM/Clang optimizations
available

Only the final link step generates
JavaScript, from bitcode

# Demo: Hello World

```
#include <stdio.h>

int
main(int argc, char **argv)
{
    printf(“Hello World!\n”);
}
```

```
% emcc –o hello.html hello.c
```

# Emscripten Installation

Installation instructions for all platforms:

**emscripten.org**

Windows installer coming very soon

- Will perform all prerequisite installation
  - Python, LLVM/clang, node.js
  - Git (for emscripten updates)

- Being developed with game developers in mind
  - Entire versioned toolchain in a single zip file

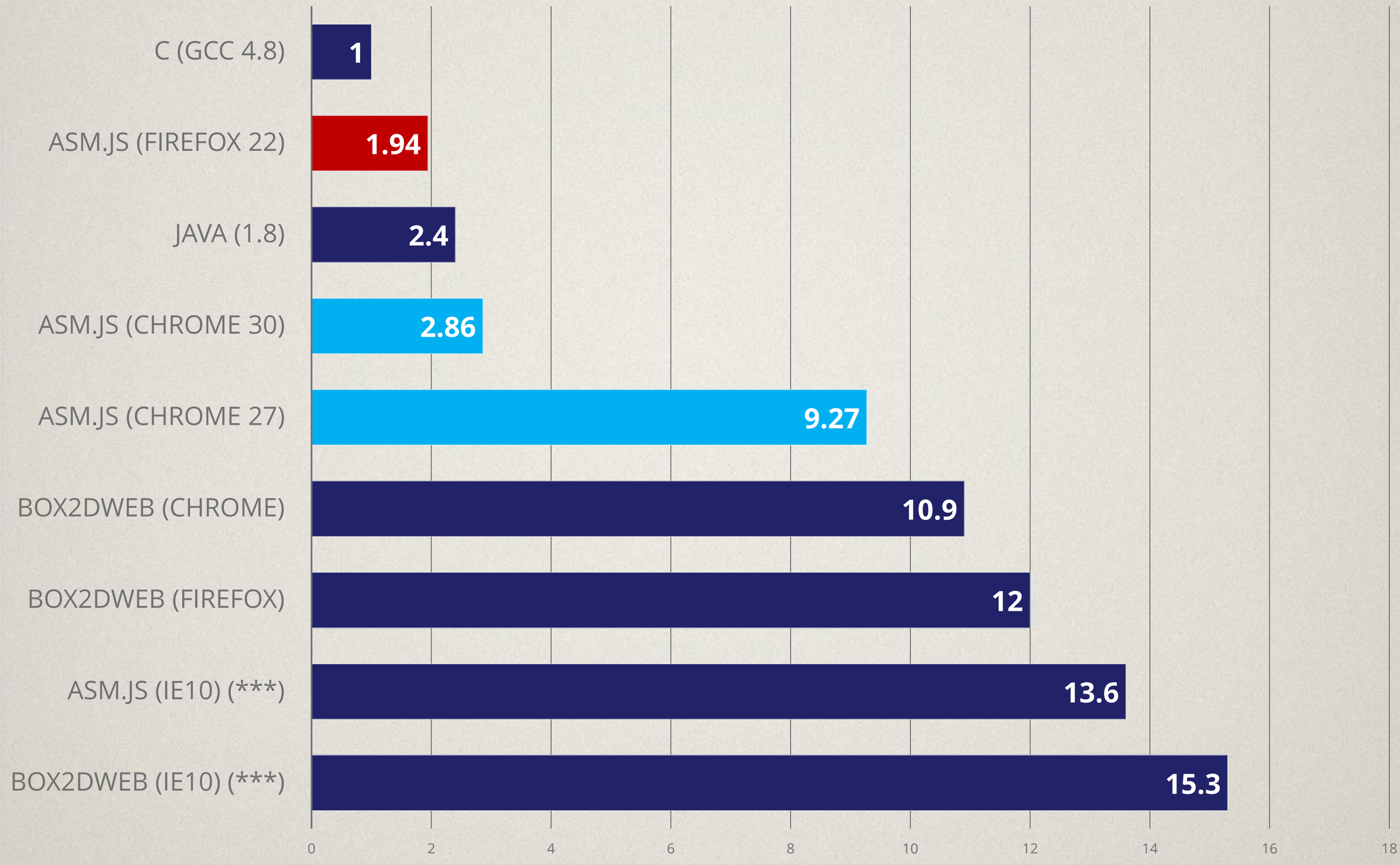
# asm.js

JavaScript subset usable as a compilation target

Formal syntax that can be specifically optimized

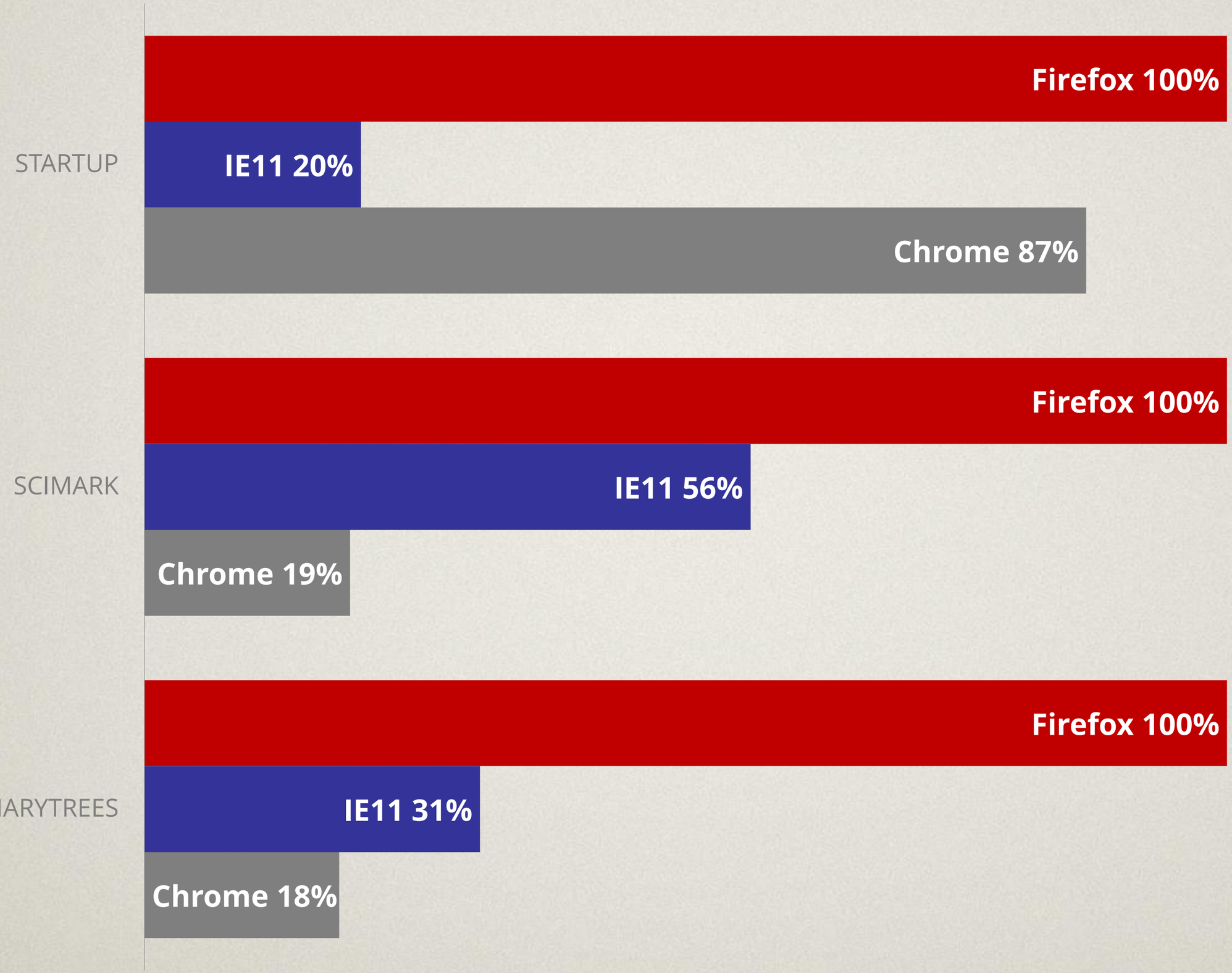
Core asm.js: functions, numbers, a heap represented by an array

# asm.js: Box2D Performance




Source: http://j15r.com/blog/2013/07/05/Box2d_Addendum

# asm.js: LuaVM Performance



# Application or Library

## C++ App

Emscripten Core

## JavaScript App

JavaScript

Emscripten Library

# The Web as a Target

Getting Started with Emscripten

# The Web as a Target

Compiler: **LLVM/Clang**

Final Stage: **Emscripten**

Libraries: **SDL, GL ES 2.0, EGL, OpenAL**

# Target Libraries

Emscripten provides emulation libraries

**SDL, GL ES 2.0, EGL, OpenAL**

(and others)

Not a perfect implementation, but usually enough

Easy to extend and add features

Libraries are not magic: regular JavaScript

```
var LibrarySDL = {
  SDL_Init: function(what) {
    document.addEventListener(“keydown”, SDL.receiveEvent);
    …
    return 0;
  }
}
```

# Development Cycle

1. Transition C++ code to use Emscripten-supported libraries (SDL, GLES, etc.)
2. Test and debug natively
3. Build with Emscripten target
4. Test and debug web port

# Build Integration

Built on **clang** & **LLVM**

Compiler looks like and replaces gcc/clang:

- gcc → emcc, g++ → em++, etc.

Compile output is LLVM bitcode

Regular LLVM “opt” step runs

Final step converts linked bitcode to JavaScript

# Graphics

OpenGL ES 2.0 is the preferred target

WebGL provides (almost) full GLES 2.0
(no client-side arrays)

Emscripten includes some “GL emulation”

Working on supporting Regal project
(more complete emulation support)

OpenGL ES 3.0 support coming soon
WebGL 2, work in Firefox in progress

# Demo: Hello OpenGL

```
#include <GL/glut.h>
#include <stdio.h>

int
main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitWindowSize(300, 300);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Browser");

    glClearColor(0, 1, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);

    glutSwapBuffers();
}
```

# The Main Loop

```
int main()
{
  while (true) {
    …
  }
```

There is no magic to handle infinite loops.

Typically one area that requires porting changes.

# The Main Loop

Solution: hand control back to Emscripten.

```
void main_loop_iteration() {
  …
}

int main() {
  emscripten_set_main_loop(main_loop_iteration,
                           0    /* use browser’s framerate */,
                           true /* simulate infinite loop */);
}
```

Loop will run once per frame.

# Demo: Hello OpenGL, v2

```
#include <GL/glut.h>
#include <stdio.h>
#include <emscripten/emscripten.h>

void draw_frame() {
    static float color = 0.0f;
    glClearColor(0, color, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);

    color = color + 0.05f;
    if (color > 1.0f) color = 0.0f;

    glutSwapBuffers();
}

int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitWindowSize(300, 300);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Browser");

    emscripten_set_main_loop(draw_frame, 0, FALSE);
}
```

# Graphics: What About 2D?

Web has Canvas 2D API

- acceleration is spotty, especially on mobile

SDL blitting API is supported on top of Canvas 2D

Best practice:

Build your own on top of GL for performance

# Audio

Simple playback through HTML5 Audio

Works well for background music

Web Audio API provides rich audio capabilities

Closest mapping on native side is OpenAL

Lots of requests for FMOD: let them know!

# Networking

Currently: UDP `sendto` and `recvfrom`

Enough to support Enet library

- Enet implements connection streams on top of UDP

Built on top of WebRTC Data Channels

Other implementations are possible, as is low level WebRTC access

# Data & Filesystem

Web has rich APIs for fetching and storing data

- all asynchronous

IndexedDB for local data storage

- no limits, prompt for >50MB in Firefox

Regular web downloads are available

- `emscripten_async_wget_data(url, arg, onsuccess, onfailure)`

Data can also be bundled via initial download

# Data: Packaging Data

Packager tool can create a virtual filesystem

Upsides:

- Use regular POSIX API (open, read, etc.)
- Easy portability

Downsides:

- Everything downloaded up front
- High memory usage

# Demo: Packaging Data

```
#include <iostream>
#include <fstream>

int
main(int argc, char **argv)
{
    std::ifstream hello_file("data/hello.txt");
    std::string data;

    getline(hello_file, data);
    std::cout << data << std::endl;
}
```

# Data: Embrace Async IO

Use custom or Emscripten APIs for asynchronously loading data from IndexedDB or the web

Upsides:

- Better interactive performance
- Lower memory usage
- Easily extended to do asset streaming

Downsides:

- Higher development complexity
- Existing code might make it difficult

# Input

Input is dependent on your library choice.

SDL: `SDL_PollEvent, SDL_PushEvent, SDL_PeepEvents, SDL_PumpEvents`

Events show up as regular SDL events

Easy testing with native SDL library

Ultimately, all events start off as web events (mousedown, mousemove, mouseup, etc.)

# Input

Web supports rich events and data gathering

- Mouse
- Keyboard
- Multitouch
- Gamepad
- Accelerometer
- Geolocation
- Microphone and Camera (via WebRTC)
- Third-party APIs (Facebook, Twitter, etc.)

All accessible via Emscripten

# Rolling Your Own Libraries

You can create your own libraries for bridging the web and your native code

For example:

- More optimized input handling
- Interaction with HTML widgets & content
- Connecting to other web packages, e.g. Facebook APIs, Persona login, Twitter integration, etc.

# Emscripten Utility API

JavaScript interaction

- emscripten_run_script
  emscripten_async_run_script

URL fetching

- emscripten_async_wget_data

Many others

- see `emscripten/emscripten.h` for more

# Demo: Utility API

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

int
main(int argc, char **argv)
{
    char *result;

    result = emscripten_run_script_string("(new Date()).toString()");
    printf("Date: %s\n", result);

    const char *script =
        "document.body.style.background = 'blue';"
        "alert('Done');";

    emscripten_run_script(script);
}
```

# Threading

Worker-style threading is possible

- No shared data
- Message passing only

Full generic threads missing currently
(but we’re working on it; code exists that we’re experimenting with)

# The Heap

The Emscripten heap is a fixed-size typed array

Maximum heap usage is currently chosen up front
- Large heaps can be difficult in 32-bit browsers
- Working on solutions in Firefox

Future optimizations:
- Resizable Heap
- Non-commit heap

# Visual Studio Integration

"vs-tool" adds Emscripten as a target

Visual Studio 2010
    2012 soon!

Compile & Run in Browser support

Windows installer for Emscripten and vs-tool coming soon

# Debugging

Debug as much as you can natively

Browser developer tools work

Code compiled with –g is readable

Working on a better native source debugging experience

# Summary of Required Changes

| | |
|---|---|
| Convert to SDL, GLES, OpenAL | • Use existing mobile code<br>• Test and develop natively |
| Convert main loop | • Make sure you can hand control back to browser |
| Package required initial data | • Ease of initial development |
| Implement async data access (if needed) | • For performance, memory usage |

mozilla

# Getting to the User

Putting your masterpiece on the web

# Deployment

## HTML driver/loader

- Page that the user navigates to
- Doesn’t need to load Emscripten JS right away
- Can do data preloads, logins, etc.

## Emscripten-compiled JS

- The .js and .js.mem file produced by Emscripten

## Any asset packages

- Assuming data loading isn’t all async

# Deployment: Code Size

JavaScript code size is similar to x86 binaries.

For a large project:

| | |
|---|---|
| Win32 binary: | 29.9 MB |
| JavaScript: | 25.1 MB |

Compressed for delivery (gzip -9):

| | |
|---|---|
| Win32 binary: | 10.1 MB |
| JavaScript: | 5.8 MB |

# Deployment: Data Delivery

"gzip" Content-Encoding is essential
    Most web servers can pre-compress,
    or at least cache compressed content

Using a CDN to deliver the data assets file is fine
    They're just files

Local caching of content in IndexedDB possible
    Caching of code coming soon

# Deployment: “Installation”

- Offline “apps” via manifests
  - Supported on Firefox for Desktop, Firefox for Android, and Firefox OS
  - Compatibility in other browsers varies
- “Packaged Apps” exist on Firefox OS
  - Single .zip file with all assets
  - Coming soon to Fx Desktop and Fx Android
- No unified packaging format (yet!)
  - In talks with others to solve this

# Making Money

Cross-platform, no-(visible)-download
- Get users playing faster
- Use as gateway

Lack of structure is powerful
- You control servers and updates
- You pick payment providers
- You choose analytics

Lack of structure is annoying
- Web Marketplaces are here/coming
- Additional infrastructure for games is coming

# What’s Coming Next?

I can see the (near) future

# Compilation Caching

**Problem**: Startup speed could be faster

**Solution**: Cache compiled JS

Much faster startup after first compile

Compilation can be done automatically, or under app control

# Apps in Workers

**Problem**: Apps all run on the main browser thread

**Solution**: Provide needed APIs on Workers

No blocking of browser content (UI or your own page)

Makes synchronous APIs possible

Full CPU utilization

Can also be solved by multiprocess browsing

We’re working on that too

# JavaScript Improvements

**Problem**: Never enough performance

**Solution**: Expose more capabilities to JS

asm.js SIMD instructions

- Likely through intrinsics
- Combined with JS “value types”

Garbage Collection hooks in asm.js

- Integrate asm.js model with JS object model

Data Parallelism in JavaScript

# More Information

emscripten.org

asmjs.org

developer.mozilla.org/games

vlad@mozilla.com