C++ on the Web: Run your big 3D game in the browser!

Quo Vadis 2013

Andre “Floh” Weissflog
Head of Development, Berlin
Bigpoint GmbH

Outline

WHY?

- Why now?
- Why C++?
- Why the web?

WHAT?

- emscripten
- flascc
- Google Native Client

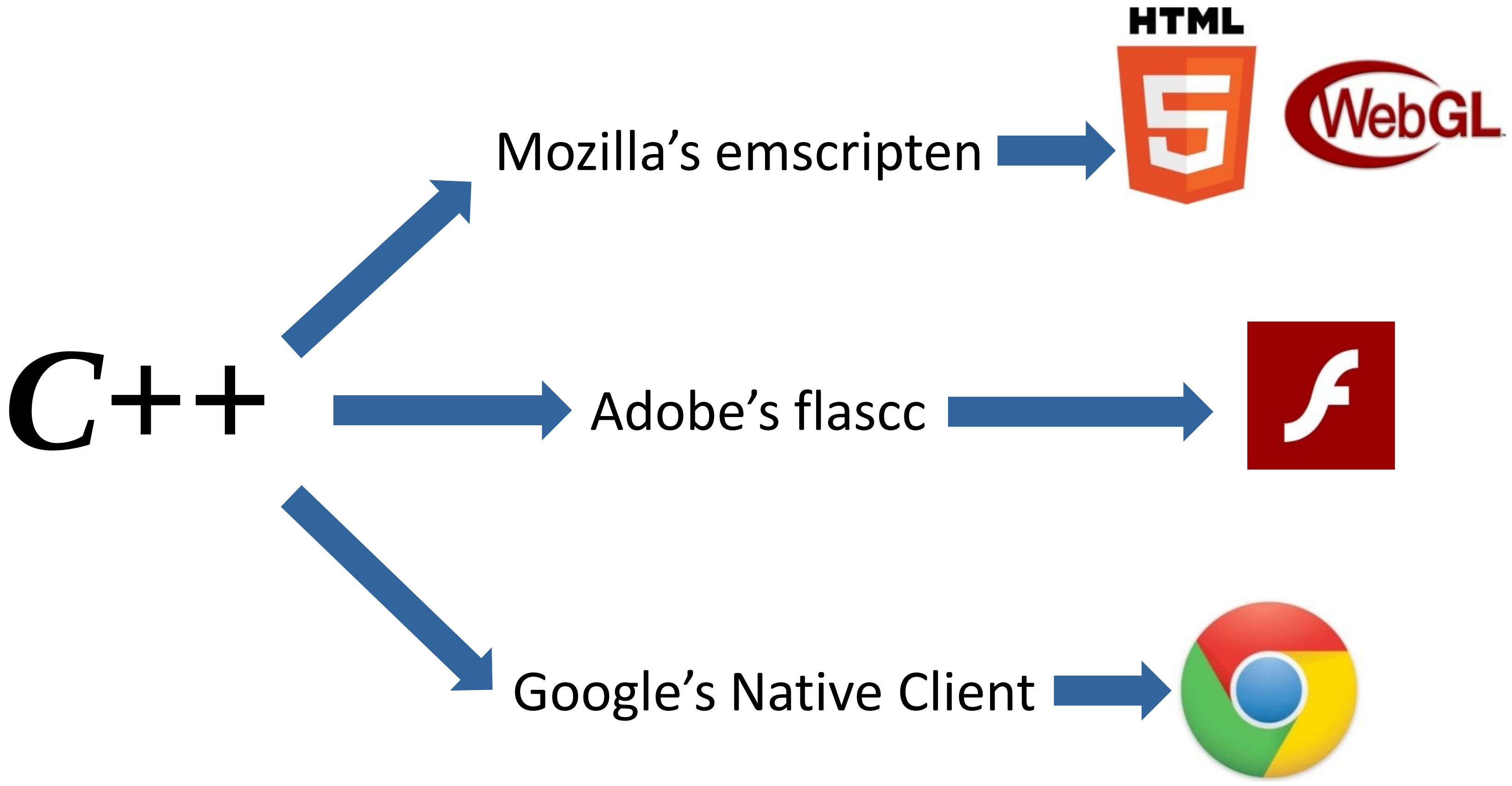
HOW?

- Build system
- Workflow & Debugging
- Specific problems and solutions

Wrap-up & Questions

Why Now?

New technologies have become mature which cross-compile C++ code to sandboxed runtime environments:



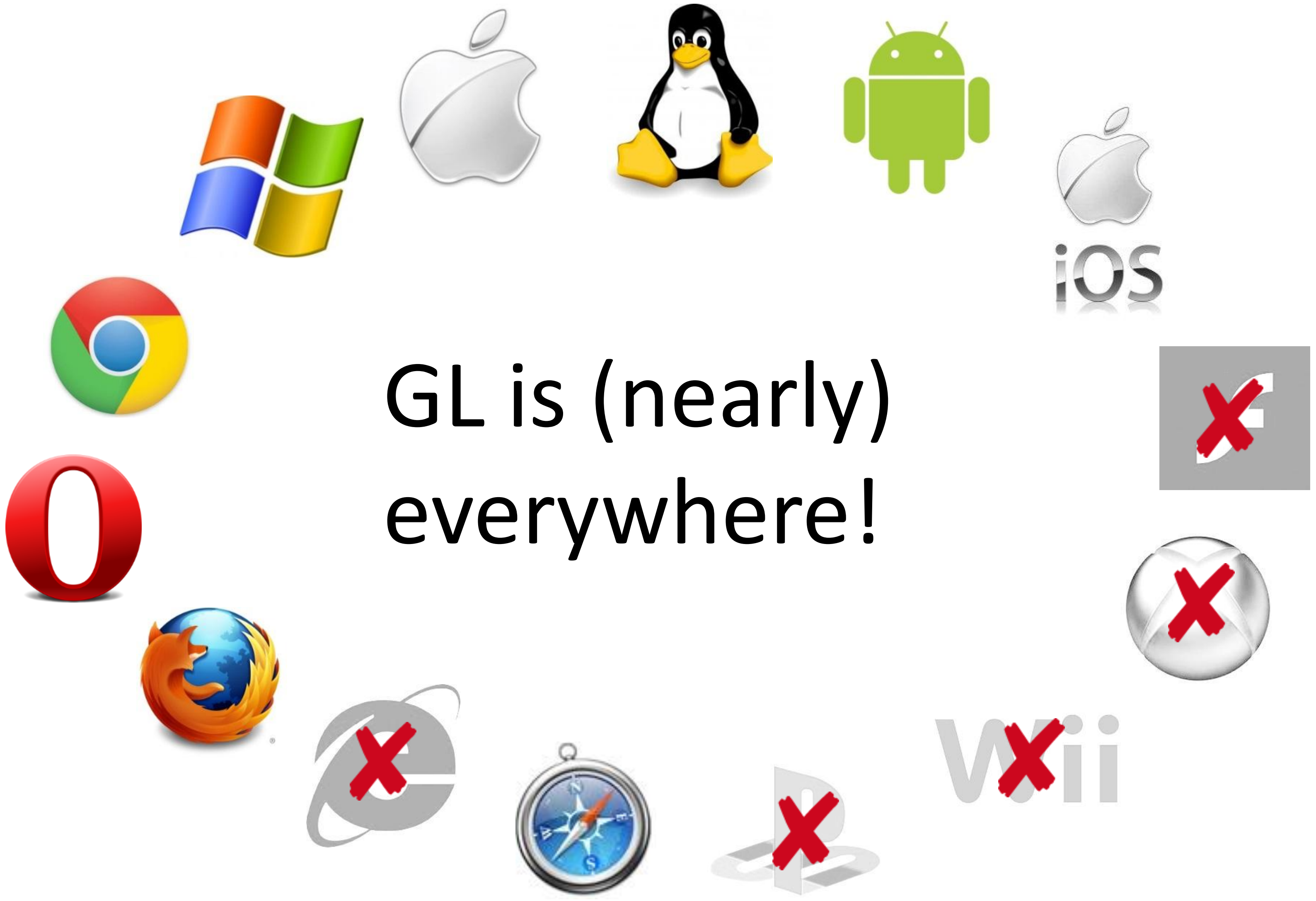
Why C++?

C++ is everywhere!

Why C++ continued…

- Support many target platforms from a single code base
- Nearly all professional game-dev middleware written in C/C++
- Good for large code-bases (static type system, compiled language, rock-solid compilers and debuggers)
- Cross-compiled C++ code is actually faster then typical handwritten Javascript or ActionScript

Why OpenGL



GL is (nearly) everywhere!

Why develop for the Web?

It’s the most friction-less platform for users **AND** developers:

Users:

- True “click-and-play”: No downloads, no installations, no patching.
- No security concerns (ideally): untrusted code runs sandboxed.

A truly open distribution platform for developers:

- no “walled gardens”
- no arbitrary certification process
- free choice of payment, hosting, “store-front” providers

No Vendor or Platform Lock-In !!

90% of your C++ code base will be platform-agnostic*

*(if done right)

Switch your target platform with a simple compile-switch!

- Consoles dying? Switch to PC!
- PC dying? Switch to mobile!
- Flash dying? Switch to HTML5!

You don’t need to “invest big” into the uncertain future of a specific platform, C++ and OpenGL got your back.

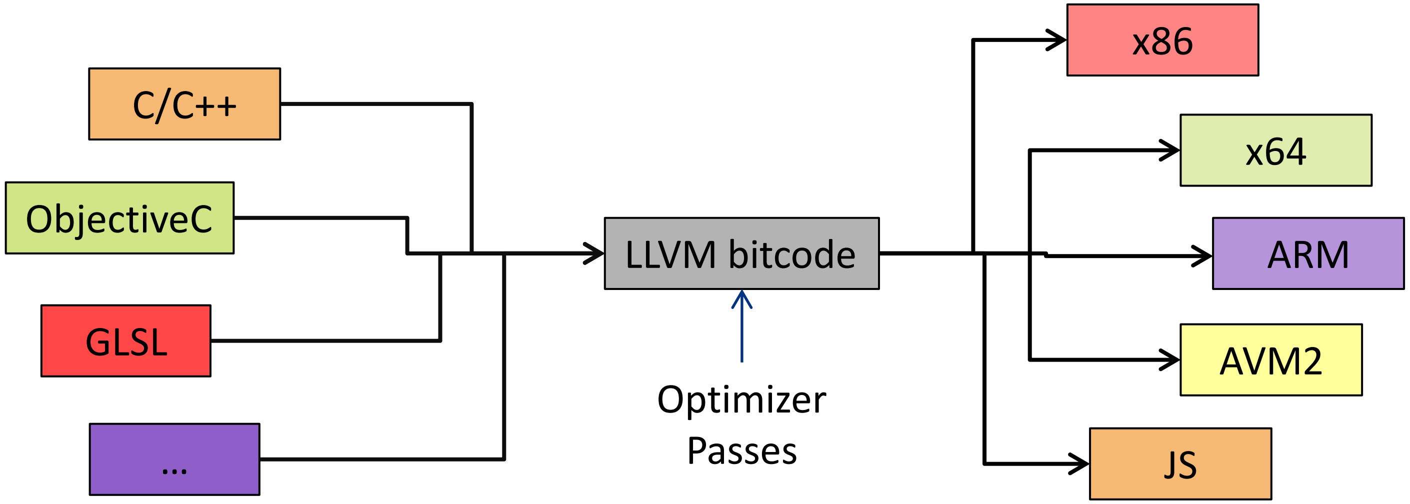
Interlude: LLVM

LLVM is the magic sauce which makes C++ on the Web possible.

A highly modular and flexible compiler infrastructure.

Started in 2000 as OpenSource project.

1. Language front-ends compile source code to “LLVM bitcode”.
2. Generic optimizer passes work on bitcode.
3. Back-ends translate bitcode to “anything” (usually CPU native instructions, but also VM byte code or different languages).



Google Native Client (NaCl)

- Project started in 2008.
- Executes pre-compiled machine code safely in a sandbox.
- Integrated into web page with the object tag (similar to Flash SWFs).
- Currently CPU-specific “executables”, pNaCl aims to solve this problem.
- Only implemented in Google Chrome.
- Currently only enabled for Chrome Web Store apps.
- Not much hope that other browsers will ever adopt NaCl.

FOR: solid runtime and tools, full CPU performance, rich API and Middleware support, full pthreads implementation

AGAINST: only on Chrome and Chrome Web Store

Adobe flascc

- Project started in 2008 as experimental “Adobe Alchemy”.
- Cross-compiles C++ code to AVM bytecode.
- Requires Flash plugin.
- Proprietary 3D API (Stage3D)
- Very slow translation of LLVM bitcode to AVM2 :(

FOR: 3D on IE, full access to Flash runtime environment

AGAINST: requires Flash plugin, extremely slow compilation process

Mozilla emscripten

- Project started in 2010.
- Cross-compiles C++ code to low level Javascript.
- Maps OpenGL to WebGL for 3D rendering.
- Rich support for standard APIs (SDL, OpenGL/ES2, EGL, GLUT, …).
- Custom WebWorker API instead of pthreads (more on that later).
- Compiler tools fast enough for large code bases.
- asm.js code generation mode (more on that later)

FOR: doesn't require plugins, runs in all browsers (except WebGL), modern JS engines "fast enough"

AGAINST: IE vs. WebGL, threaded code can't simply be cross-compiled

emscripten FTW!

Will concentrate on emscripten from here on.

Why? (all IMHO of course):

- Everything about emscripten is completely open.
- Biggest “near future potential”.
- Extremely responsive and fast development team.
- Seamless integration into web page.
- It’s “fast enough” already, and getting still faster.
- asm.js!

BUT: all 3 technologies are very similar to port to.

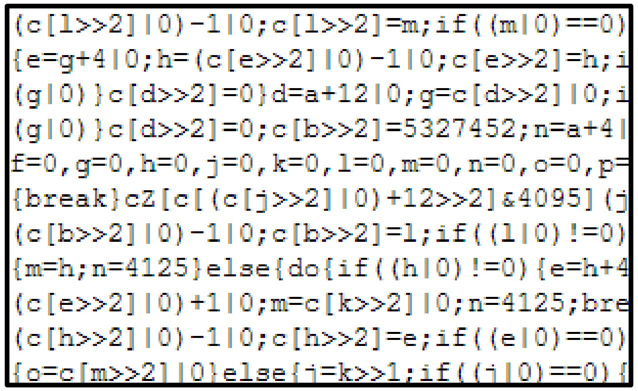
Why is emscripten-generated JS fast?

emscripten is **NOT** a source-level translator from C++ to high-level JS!

Instead it translates LLVM bit code to Javascript which looks like “high-level assembly”:

$$\begin{tikzcd} \text{C++} \arrow[r] & \boxed{\textit{LLVM}} \arrow[r] & \text{.bc} \arrow[r] & \boxed{\textit{emscripten}} \arrow[r] & \text{.js} \end{tikzcd}$$

emscripten uses a big TypedArray as C/C++ memory heap.



```
(c[l>>2]|0)-1|0;c[l>>2]=m;if((m|0)==0)
{e=g+4|0;h=(c[e>>2]|0)-1|0;c[e>>2]=h;i
(g|0)}c[d>>2]=0}d=a+12|0;g=c[d>>2]|0;i
(g|0)}c[d>>2]=0;c[b>>2]=5327452;n=a+4|
f=0,g=0,h=0,j=0,k=0,l=0,m=0,n=0,o=0,p=
{break}cZ[c[(c[j>>2]|0)+12>>2]&4095](j
(c[b>>2]|0)-1|0;c[b>>2]=l;if((l|0)!=0)
{m=h;n=4125}else{do{if((h|0)!=0){e=h+4
(c[e>>2]|0)+1|0;m=c[k>>2]|0;n=4125;bre
(c[h>>2]|0)-1|0;c[h>>2]=e;if((e|0)==0)
{o=c[m>>2]|0}else{j=k>>1;if((j|0)==0){
```

Emscripten generated JS (minified)

RESULT:

- no objects, no properties
- no dynamic typing
- ***just functions, numbers and local variables***
- no garbage collection!
- LLVM optimization passes!

What's up with ***asm.js***?

From the spec: a strict subset of JavaScript that can be used as a low-level, efficient target language for compilers.

Formalizes the subset of JS generated by emscripten and similar JavaScript generators, BUT is still **standard Javascript**.

Good performance in standard JS engines, big performance boost in asm.js aware JS engines (e.g. in Firefox Nightly).

How fast?

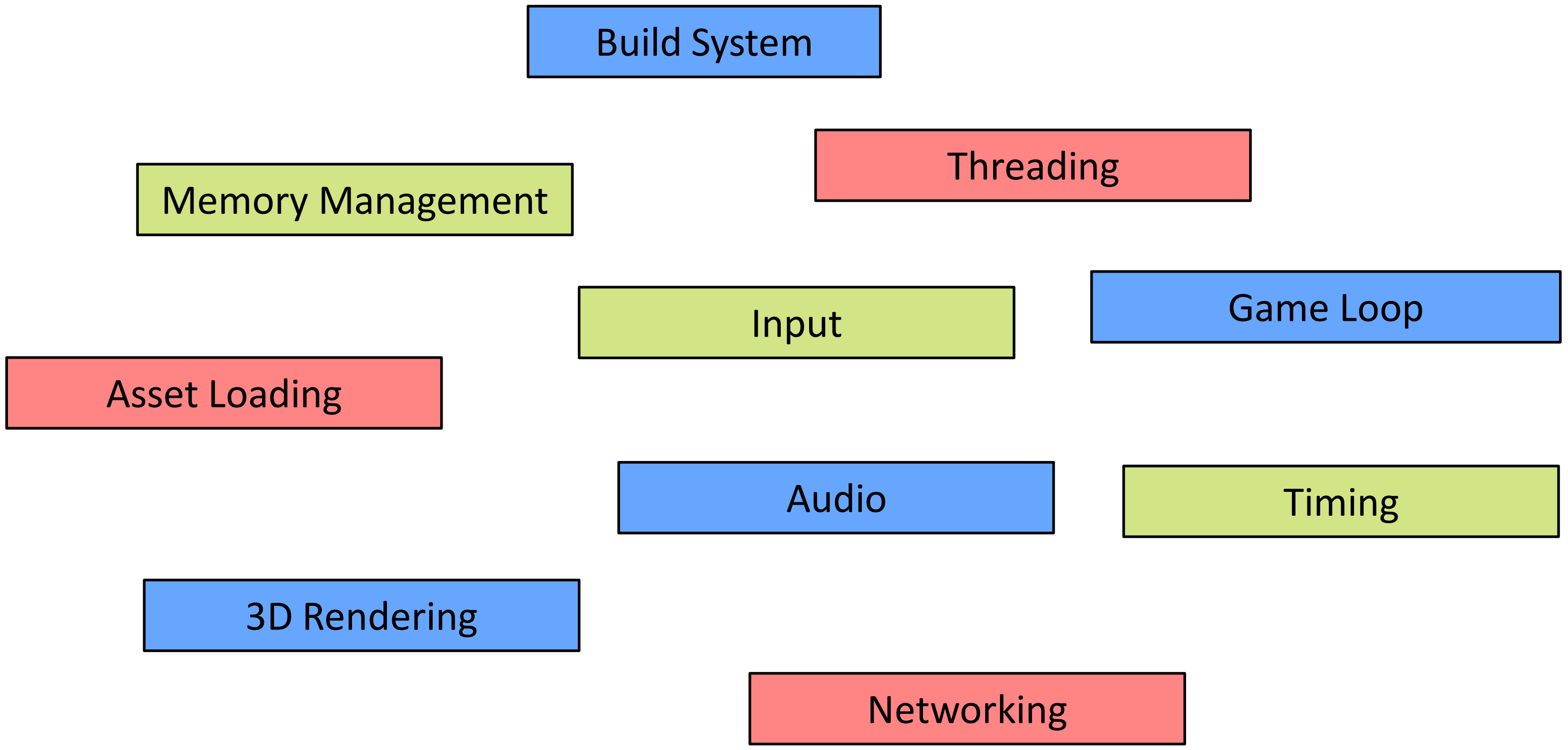
Within **2x** of native code (gcc –O2), and performance is much more consistent.

But no matter whether asm.js is specifically optimized by an JS engine:

If your game is GPU bound, CPU performance isn't really that important.

In our experiments, "CPU speed" is fast enough even without asm.js, it's WebGL and the GPU that really matters.

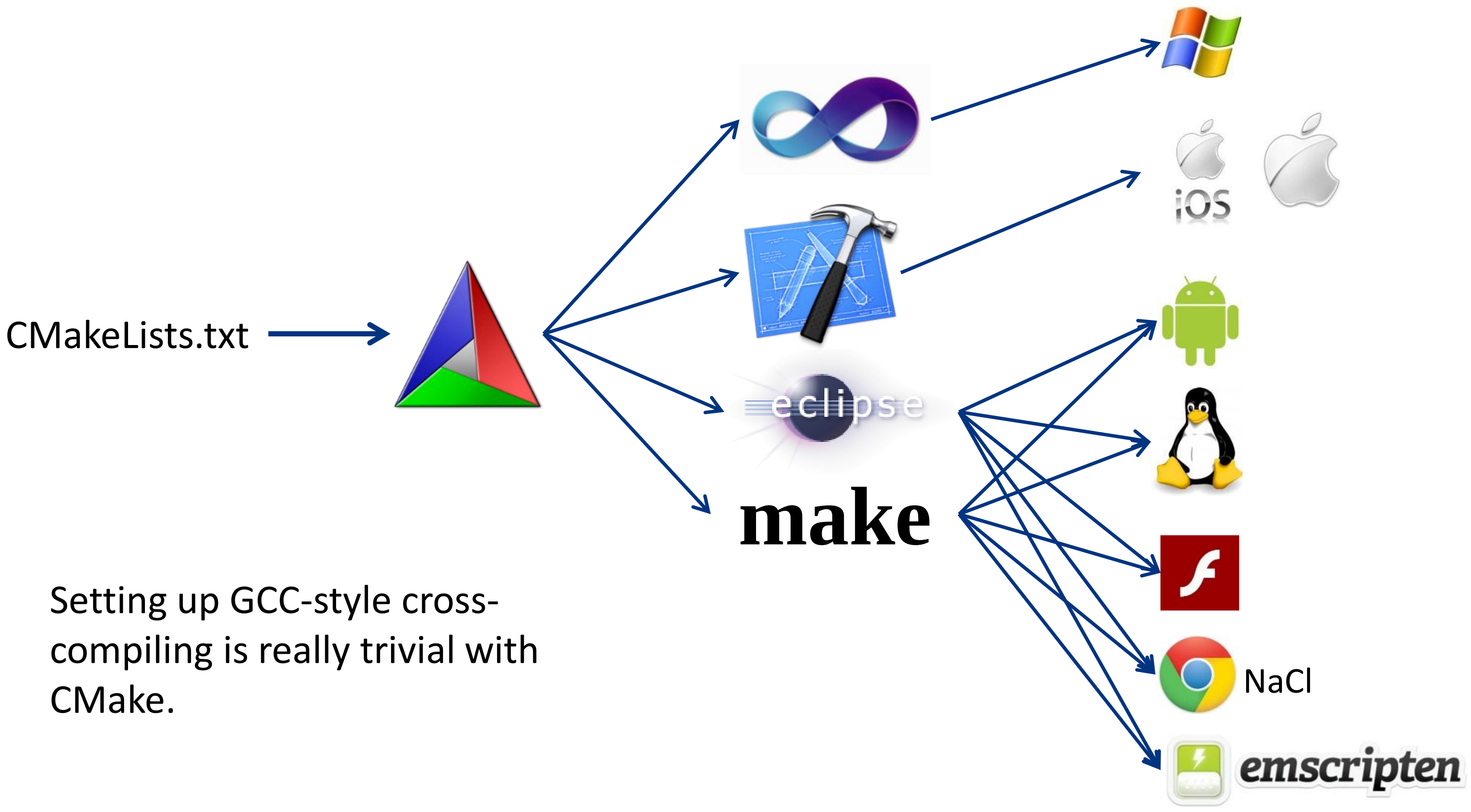
Platform-dependent components of a 3D Game Engine



Let's look at those one by one (in no particular order)

Interlude: CMake

Cmake is a **meta-build-system**: create build files for many platforms and IDEs from a single set of project description files (CMakeLists.txt)



Setting up GCC-style cross-compiling is really trivial with CMake.

emscripten's Build System

What ***emscripten*** provides:

- gcc compatible drop-in tools (emcc, em++, emar, …)
- system headers
- system libs
- hello_world.cpp, and lots of other samples

What ***you*** provide:

- Makefiles which invoke ***emcc*** instead of ***gcc*** (easily generated with a "cmake toolchain file")

Only 3 things are important in a cross-compiling scenario:

- The right compiler is called.
- The right headers are included.
- The right libs are linked.

Make sure that the cross-compiling SDK's compiler, headers and libs are used, not the host system's native files!

emscripten is very fool-proof about using the right headers and linking against the right libs.

BIGPOINT

Workflow & Debugging Tips

Make sure your app compiles and runs also as normal OSX or Windows app!

Do your normal development work and debugging on this native desktop version inside XCode or Visual Studio, this gives you the best turnaround times and debugging experience.

Only fallback to “low level debugging” for bugs in emscripten-specific code.

Having said that, low-level debugging of emscripten-generated JS isn’t that bad:

```
function __ZNK9Graphics215CameraComponent14GetViewFrustumEv($this)
{
        var label = 0;
        var $1;
        $1=$this;
        var $2=$1;
        var $3=(($2+12)|0);
        var $4=__ZNK6Shared14CameraSettings14GetViewFrustumEv($3);
        return $4;
}
```

Inject log messages directly into JS without recompiling, make use of the browser’s Javascript debugger and profiler, etc…

emscripten Memory Management

emscripten uses a single, big TypedArray with multiple views as heap for the C/C++ code:

```
var buffer  = new ArrayBuffer(TOTAL_MEMORY);
HEAP8       = new Int8Array(buffer);
HEAP16      = new Int16Array(buffer);
HEAP32      = new Int32Array(buffer);
HEAPU8      = new Uint8Array(buffer);
HEAPU16     = new Uint16Array(buffer);
HEAPU32     = new Uint32Array(buffer);
HEAPF32     = new Float32Array(buffer);
HEAPF64     = new Float64Array(buffer);
```

Pointers are actually indices into this typed array

emscripten links against dlmalloc.c to for malloc()/free()

From a C/C++ programmer's perspective, dynamic memory management isn't different from other platforms. Just don't spam malloc()/free() and be aware of memory fragmentation.

Input Handling and Timing

EASY!

emscripten offers mouse / keyboard input support through the ***GLUT, GLFW*** and ***SDL*** library wrappers.

If your code already sits on top of GLUT, GLFW or SDL, you don't need to do anything.

Otherwise it's trivial to add minimal wrapper code interfacing those APIs just for input.

Nebula3 currently uses GLUT code to setup the display and handle input, the emscripten team recommends to use SDL though, since it is used more broadly.

For timing, emscripten has a simple function **emscripten_get_now()** which returns a float number in milliseconds.

Break up your Game Loop!

Your Javascript code runs inside a per-frame callback during the “browser loop”, and there’s no way around it.

Spending too much time in your code will affect the whole browser (sluggish behaviour, freezes, eventually the browser will kill your page).

YOU CANNOT OWN THE GAME LOOP!

NEVER BLOCK FOR MORE THEN A FEW DOZEN MILLISECONDS

You may have to rethink your high-level application model to solve this problem. Always return after XX milliseconds to the browser, and split expensive work into pieces, and/or move to WebWorker threads!

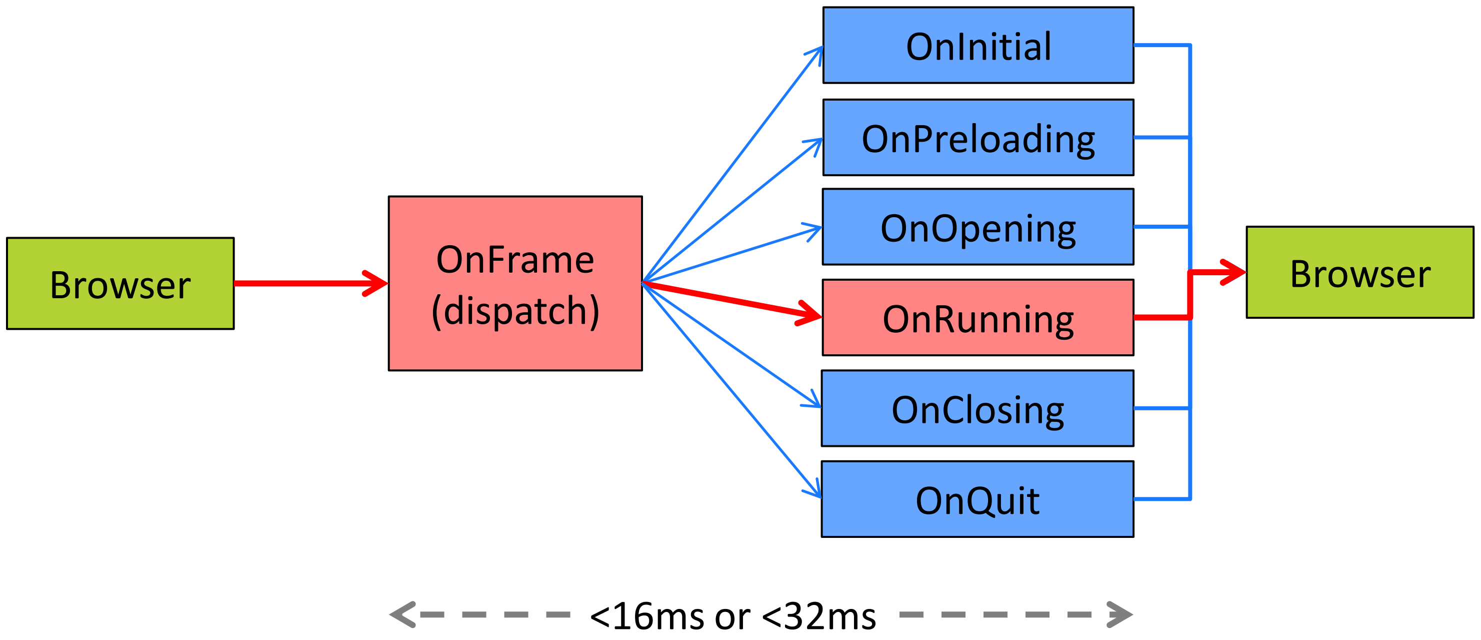
The Flash guys had to solve the same problem many years ago, google for ***“Elastic Racetrack”*** and ***“Marshaled Slices”.***

Break up your Game Loop, Nebula3 style

Nebula3 uses a “PhasedApplication” model, app goes through states/phases which tick themselves forward when phase is finished.

Individual per-frame-callbacks should return within 16 or 32 milliseconds, and never block.

Expensive (init) work is distributed across several frames.



Multithreading / Parallelism

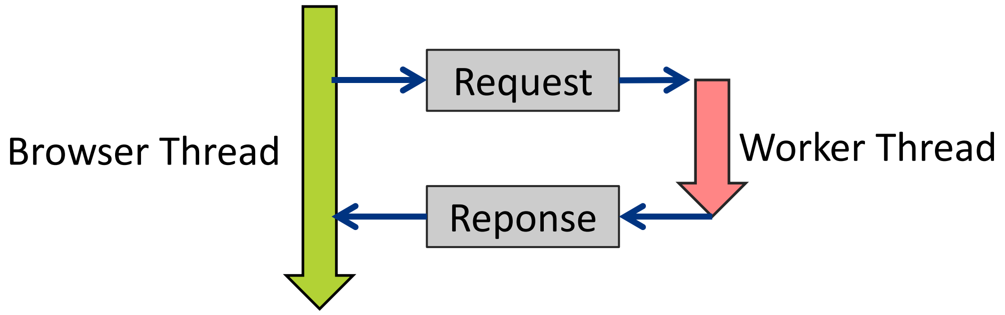
This can be tricky since emscripten doesn't support pthreads.

OMG WHY? *...because JavaScript.*

JS doesn't allow shared state between "threads".

WebWorkers to the rescue!

WebWorkers implement a simple, high level parallel-task system.



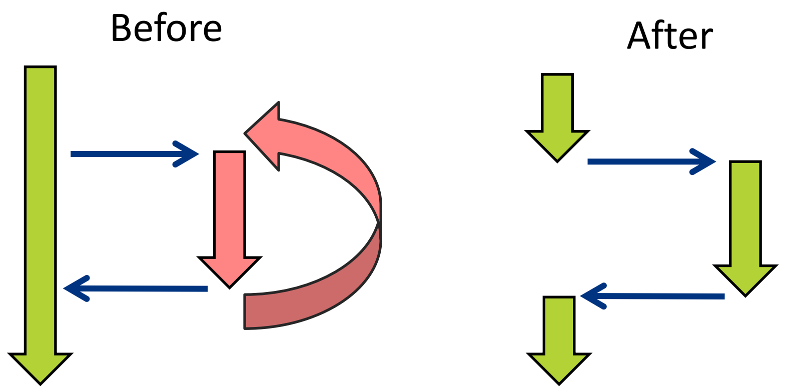
Good thing about WebWorkers: it's really hard to shoot yourself in the foot.

Because: no mutexes, no crit-sects, no cond-vars, no atomic-ops…

Multithreading porting strategies

So what about your existing multithreading code?

Option 1: slice it into frames and call an UpdateThread() method from main loop:



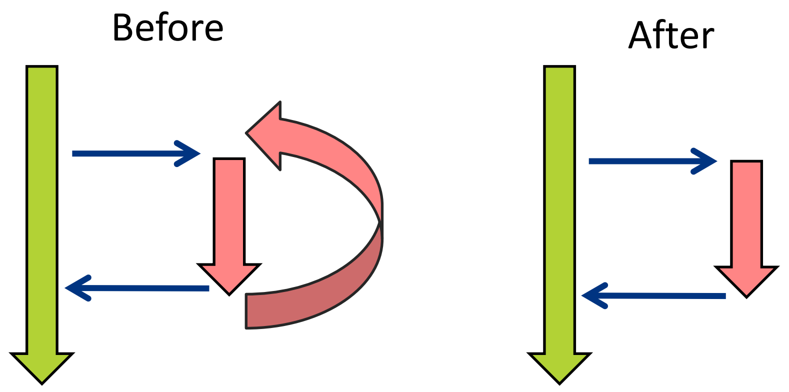
No multithreading, everything runs on the browser thread now ☹

But useful to get your code up and running quickly.

Only option if your code absolutely relies on shared state.

Multithreading porting strategies

Option 2: move to web workers (preferred)



Watch the overhead of getting data in/out of web worker!

Worker code must reside in its own .js, and has its own address space, so don't pass pointers around.

Workers have limited runtime environment (no DOM access, no GL, …

Asset Loading

There's ***The Easy Way***, and ***The Right Way*** to load assets.

The Easy Way

1. Preload all files from HTTP server and map them into "virtual file system" in RAM.
2. Use fopen / fread / fclose to read data synchronously.

emscripten makes this approach extremely easy, and you don't need to change your existing loader code.

But (big BUT):

Big monolithic asset download required before your app can start.
Not very user friendly, and may be impossible with big games.

Asset Loading

The Right Way

Implement "HTTP file system" which asynchronously downloads files from web server on demand.

Engine must be able to handle asynchronous loading for all file types: up to several seconds until data is available.

Can't simply block until data is downloaded! The whole browser will block too and eventually kill the page.

Just think of the web as an ***extremely slow and unreliable*** HDD or DVD.

emscripten has several C wrapper functions to start downloading data from an URL, with callbacks when succeeded / failed, uses XMLHttpRequest inside.

Networking / Multiplayer

This is the bleeding edge

Not much personal experience with this yet, but some pointers:

WebSockets are widely available, but sorta suck (esp. for action games).

WebRTC data channels offer UDP- and TCP-based, peer-to-peer messaging. Much better but **not yet standardized** across browsers.

emscripten wraps both WebSockets and WebRTC, and has a working ENET port (http://enet.bespin.org/).

Audio

This is another area full of pain:

<audio> tag** vs **WebAudio

The ***<audio> tag*** is fundamentally broken and hopefully dies a painful death:

- Browsers either support MP3 or Ogg Vorbis, but not both
- Designed for simple web pages, but not demanding 3D games.

WebAudio is (hopefully) the future, but currently only supported in Chrome and Firefox Nightly.

Web Apps will have to rely on ugly hacks and workarounds for the foreseeable future.

emscripten has SDL Audio (based on <audio> tag), and OpenAL (based on WebAudio).

3D Rendering

OpenGL ES2 support is really good in emscripten. If you code is already running on mobile platforms, you’re ready to go.

GL context setup via GLUT, SDL, GLFW or EGL, your choice.

DXT texture format transparently exposed through extension, but beware of PowerVR on mobile platforms!

WebGL has higher function call overhead than native OpenGL, so reducing number of GL calls is important.

In Firefox and Chrome on Windows, WebGL is actually translated to Direct3D9 (surprisingly fast!).

Wrap-up

- You can run big C/C++ code bases (“a million lines of code”) in the browser.
- Javascript is already fast enough for many types of games.
- Massive performance improvements happening right now (better code generation, JS engines better at running generated code, asm.js…)
- Development moves fast in the browser world, but standardization is slow.
- Audio and Networking in the browser is still a pain.
- WebGL is here to stay.

Tech Blog: http://flohofwoe.blogspot.com
Demos: http://www.flohofwoe.net/demos.html

BIGPOINT

Questions?

Thanks!

Bigpoint GmbH
Andre Weissflog
Head of Development, Berlin

Drehbahn 47-48
20354 Hamburg
Germany

Tel +49 40.88 14 13 - 0
Fax +49 40.88 14 13 - 11

info@bigpoint.net
www.bigpoint.net

Find us on