https://www.beautiful.ai/presenter

WebAssembly

WebAssembly (a.k.a. wasm) an efficient, low-level bytecode for the web.

Oded Soffrin
FED @ Wix Photographers

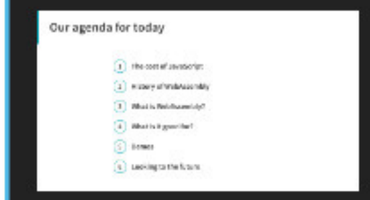
Noam Neeman
FED @ Wix Answers

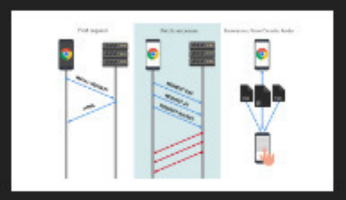
11:44:43 AM

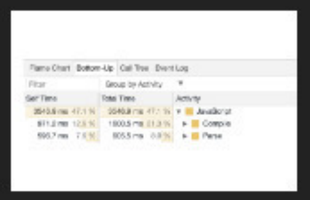
1/50

00:00:36

-introduce ourselves. talk about web assembly and why we will still have jobs as js devs.







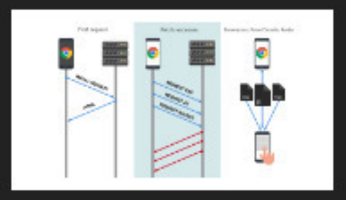
Our agenda for today

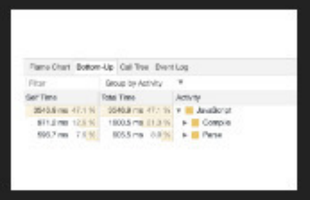
1. The cost of JavaScript
2. History of WebAssembly
3. What is WebAssembly?
4. What is it good for?
5. Demos
6. Looking to the future

11:45:22 AM

00:00:17

It doesn't make a lot of sense to speak about WebAssembly without going through the history of JS. so first we'll run through the fundamentals of JS engines, then will go over some of the history that made the riverbed that led to the creation of webAssembly. then we'll then we'll dive into web assembly itself, following with some exciting live demo's and finally a look to the future.





Section 1

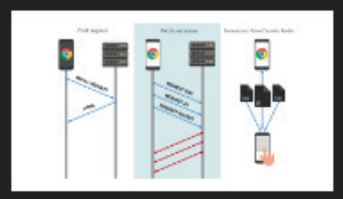
The cost of JavaScript

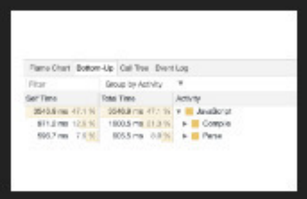
11:45:33 AM

3/50

00:00:00

No notes





https://www.beautiful.ai/presenter

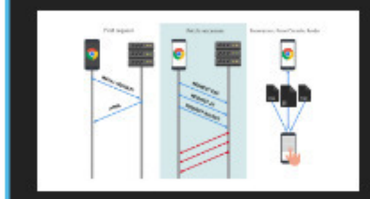
Addy Osmani Following
Eng. Manager at Google working on Chrome • Passionate about making the web fast.
Nov 15, 2017 · 8 min read

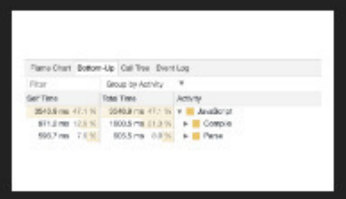
11:45:54 AM

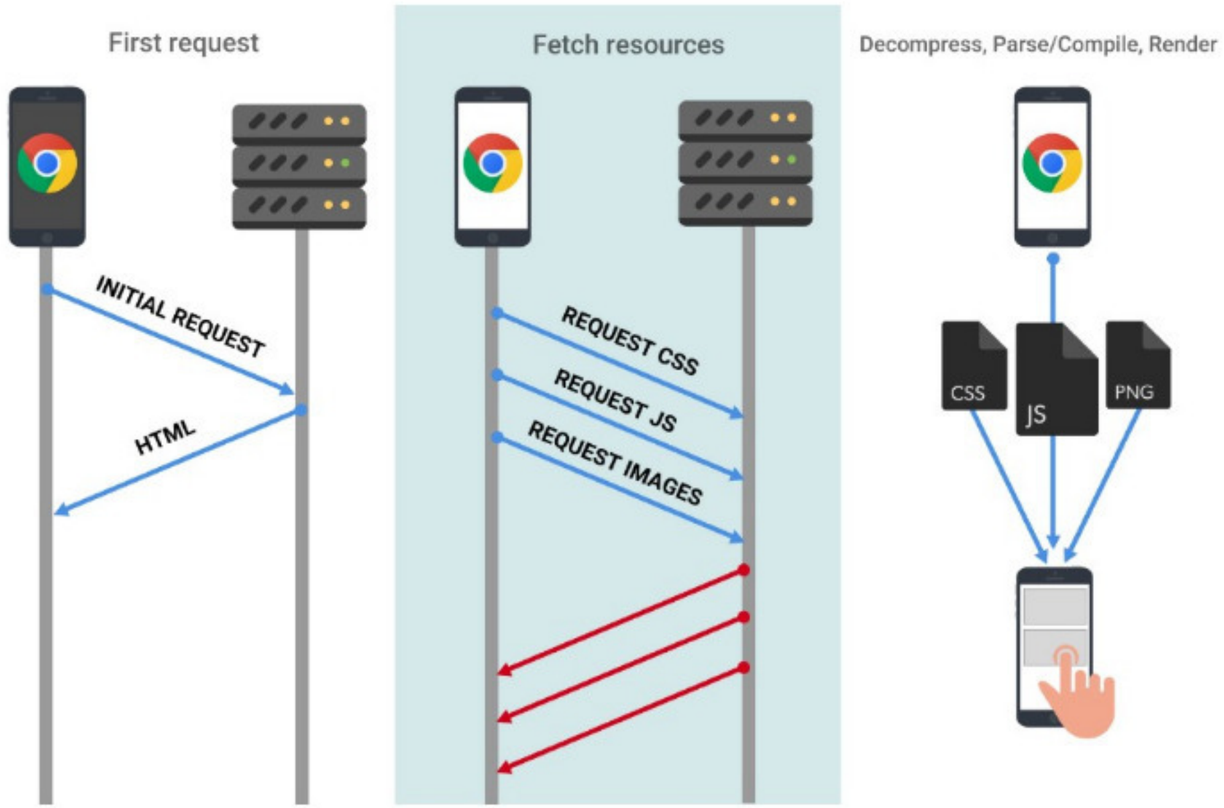
4/50

00:00:15

There's an article on medium called "the cost of js" by a Google Chrome Eng - Addy Osmani. Must Read. some of the slides that follow are exerts from his article







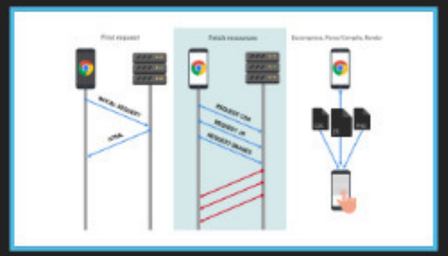
11:46:03 AM

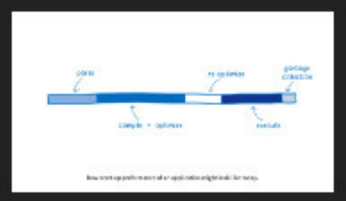
5/50

00:00:04

Let's start from the very basics This is an outline of how we get our apps running in the browser When most developers think about the cost of JavaScript, they think about it in terms of the download & execution cost. We do all these tricks to minimise the packages that are sent.






Minify _everything_
Babelified ES5 w/Uglify
ES2015+ with babel-minify
css-loader + minimize:true

Transpile less code
babel-preset-env + modules:false
Browserlist
useBuiltIns: true

Code-splitting
Dynamic import()
Route-based chunking

Scope Hoisting:
Webpack 3
RollUp

Tree-shaking
Webpack 2+ with Uglify
RollUp
DCE w/ Closure Compiler

Strip unused Lodash modules
lodash-webpack-plugin
babel-plugin-lodash

Optimize "Vendor" libs
NODE_ENV=production
CommonsChunk + HashedModuleIdsPlugin()

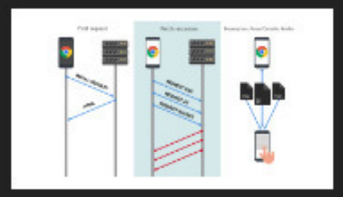
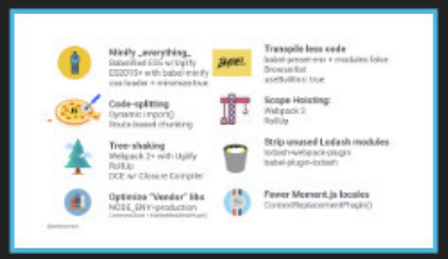
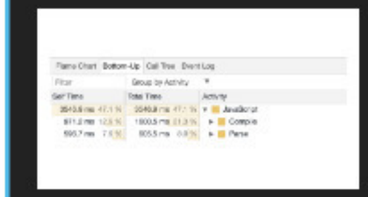
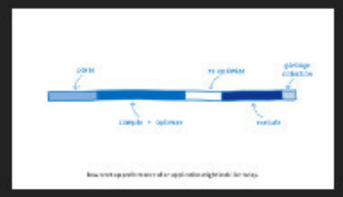
Fewer Moment.js locales
ContextReplacementPlugin()

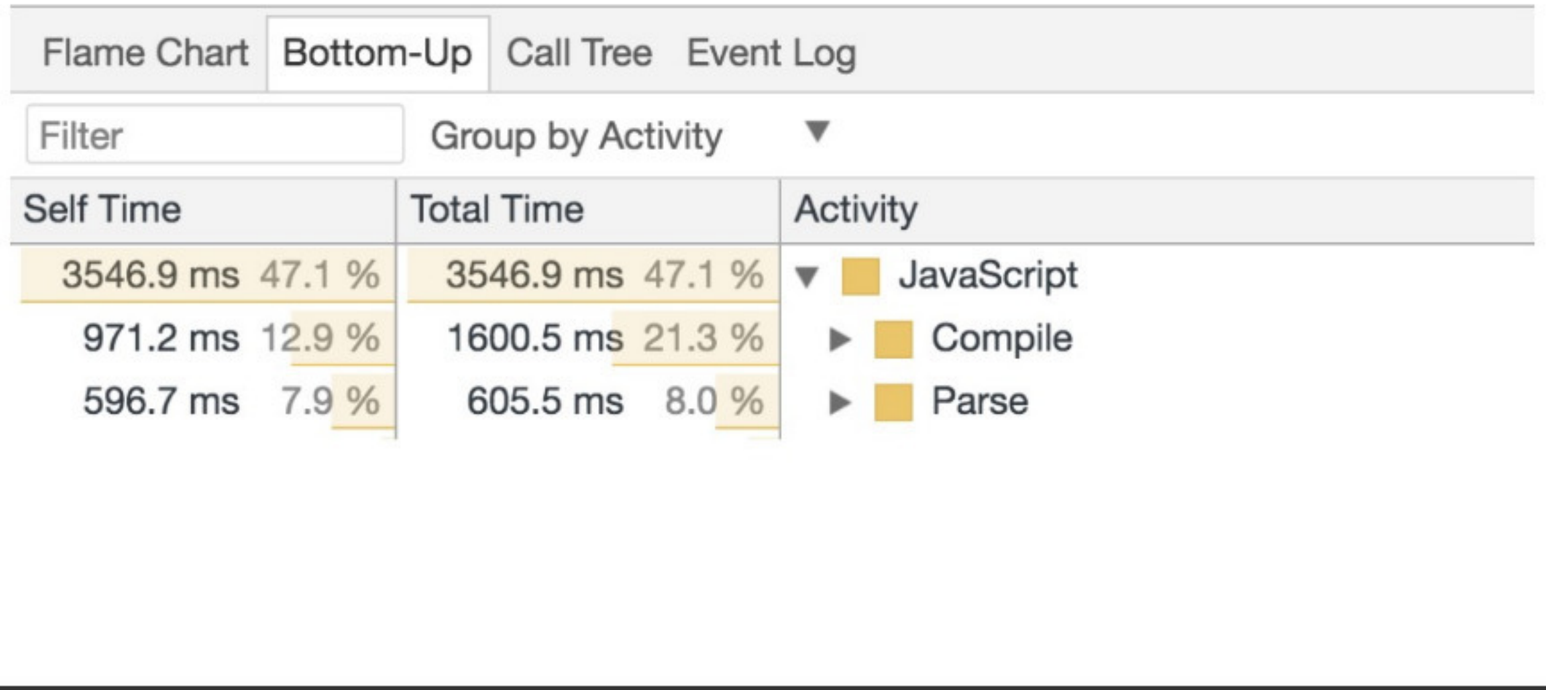
@addyosmani

11:46:14 AM 6/50 00:00:02

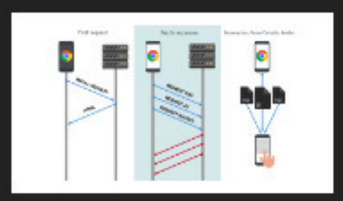
We do all these and more, to reduce bundle size and transfer time. -Code-splitting - only ship needed code - Minify -Removing unused code -Caching - minimise network trips what's usually goes unthought of and forgotten, parse and compile time have you ever looked at this?

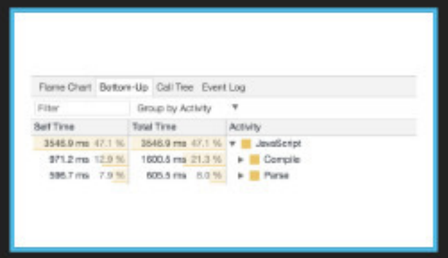


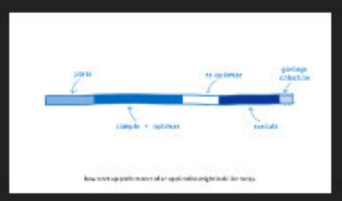
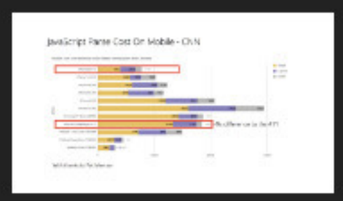





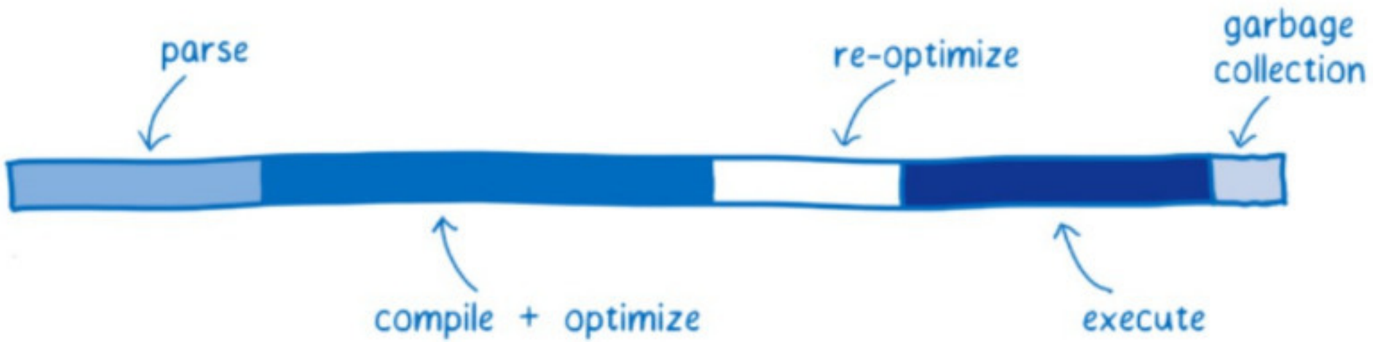


BTW You can see these stats under Bottom-Up in the performance tab in chrome dev tools. Parsing the time it takes to process the source code into something that the interpreter can run. AST - Abstract Syntax Tree Compiling - the time that is spent in the compiler. Some of the "optimizing compiler"'s work is not on the main thread, so it is not included here.



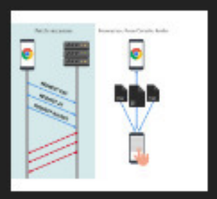



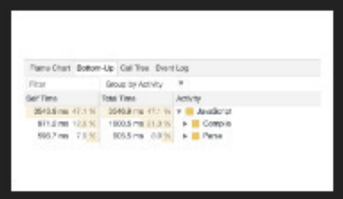
how start-up performance of an application might look like today.

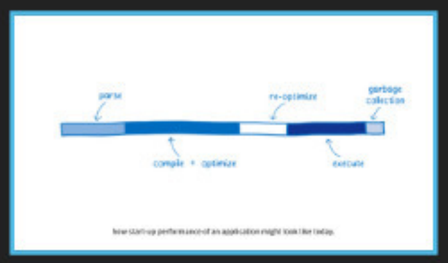
11:46:28 AM

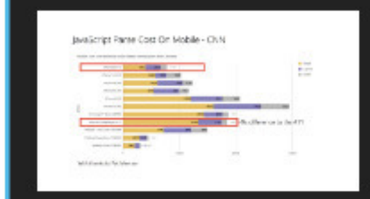
00:00:00

just to recap, these are the steps we take parse optimizing Re-optimizing Execution Garbage collection all of this happens after the resource has been downloaded.

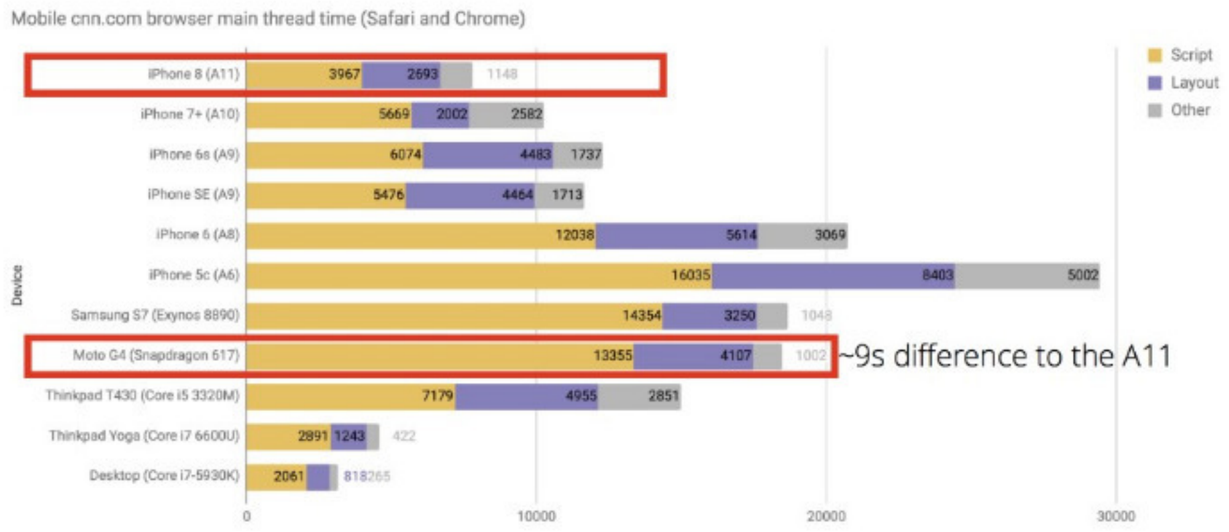








JavaScript Parse Cost On Mobile - CNN



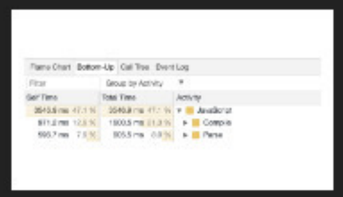
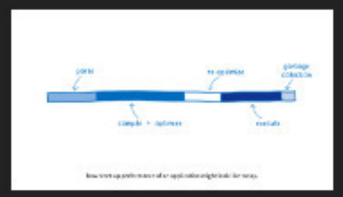
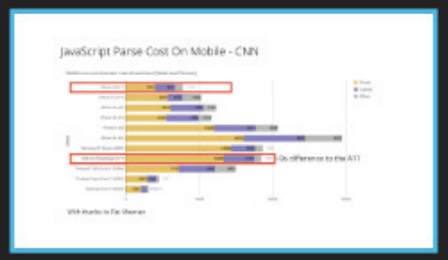
With thanks to Pat Meenan

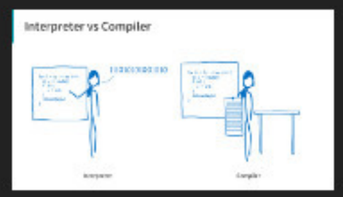
11:46:32 AM

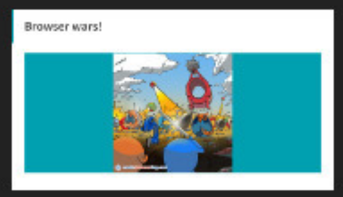
9/50

00:00:00

just for comparison, on mobile: CNN.com takes ~4s to parse on an iPhone 8 and up to 13s on a Moto G4.





https://www.beautiful.ai/presenter

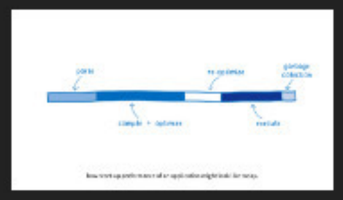
A crash course in just-in-time (JIT) compilers

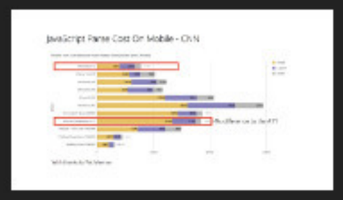
By Lin Clark

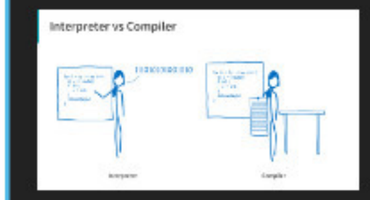
Posted on February 28, 2017 in A cartoon intro to WebAssembly, JavaScript, Performance, and WebAssembly ❤ Share This

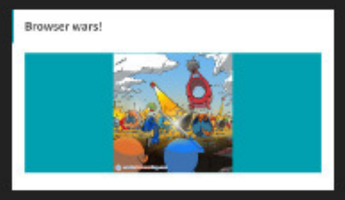
11:46:36 AM 10/50 00:00:00

Now that we know that the cost of parsing is high, we can go on to the next step, compiling time, but wait, compiling? should javascript be interpreted? who here ever heard of JIT? great article by Lin clark, some of the slides including graphics are exerts from her series of articles about web assembly.

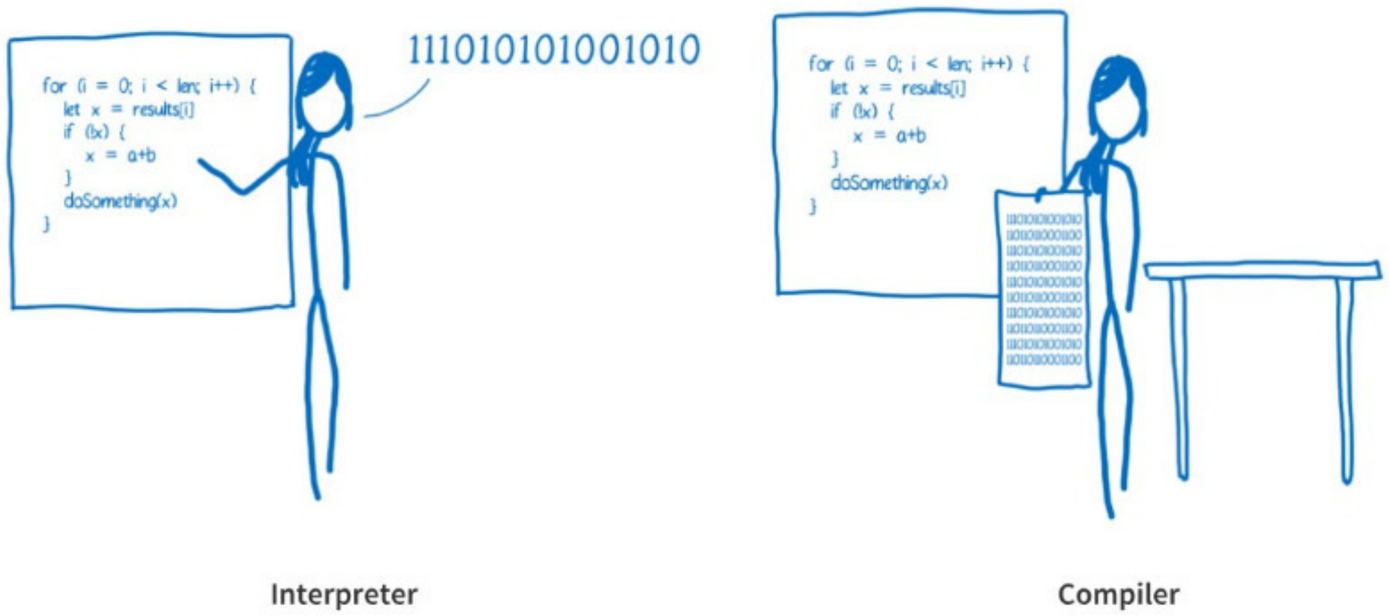








Interpreter vs Compiler



Interpreter

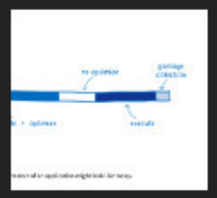
Compiler

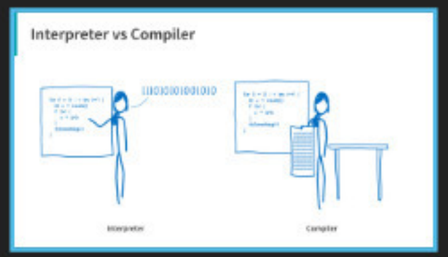
11:46:40 AM

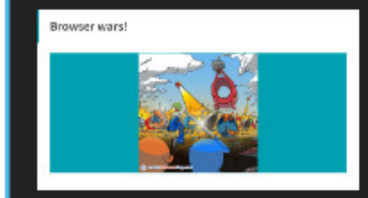
11/50

00:00:00

now that we have parsed code (AST) we need to run it, but machines only understand machine language, instructions which can actually run on the processor originally there were two ways of doing this. An interpreter does this line by line, going over every line of code (AST) every time it's hit. and compiles it on the fly to a machine instructions. on the other hand compiler takes all the code in at once and spits out the executable code. can take its time doing so for various verifications and optimisations. the compiler compiles code for a specific hardware architecture. and the code will only run on that specific architecture. seems like it's obvious why initially interpreters seemed like a really good fit for the web. code could transfer over the network then up and running fairly quick, no matter what hardware was on the client end. Up to 2008, browsers actually used a plain old interpreter to run JS. the big disadvantage was brought to light as heavier and heavier apps were ran. a new technology was needed to solve this. in came JIT compiler - just in time

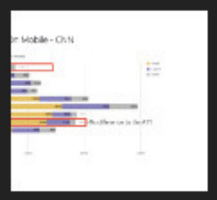


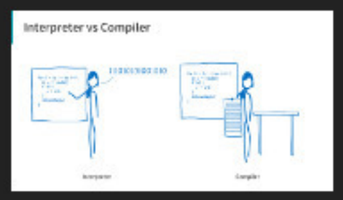



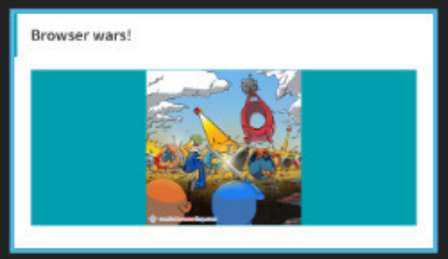


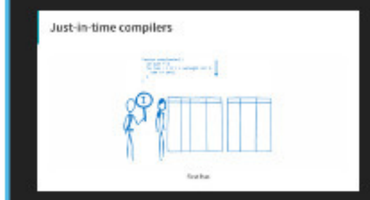
Browser wars!

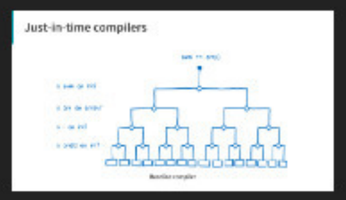
as of 2008, fuelled by browser wars. Browser companies looked for various ways to make their browsers fast. that meant running site code faster.



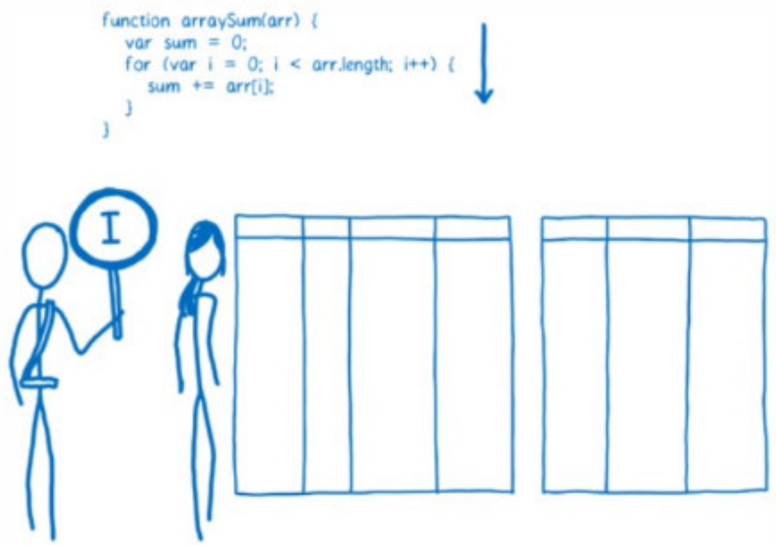







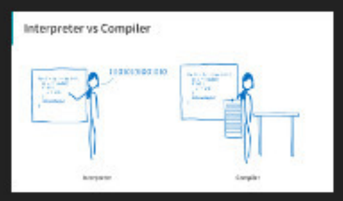


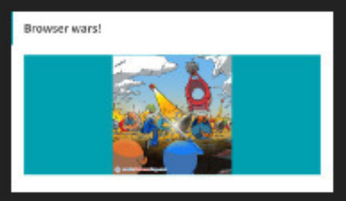
Just-in-time compilers

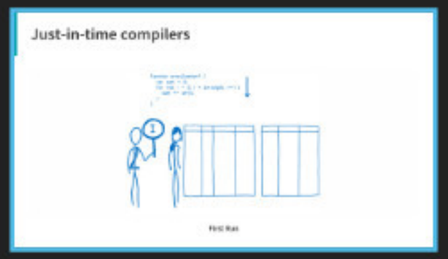


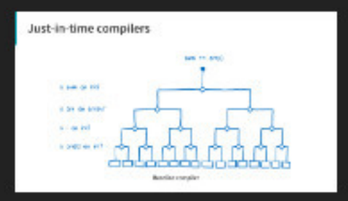
First Run

that's when JIT compilers were first introduced in the browsers JS engines. JIT was not a new concept, it dates back to 1970's over LISP. Every browser did it differently. but the underlying concept is the same. JIT uses the same interpreter as before mentioned but add a crucial component the PROFILER or MONITOR. the monitor surveys the execution frequency of each line that runs. the first time around, running normal interpreter.

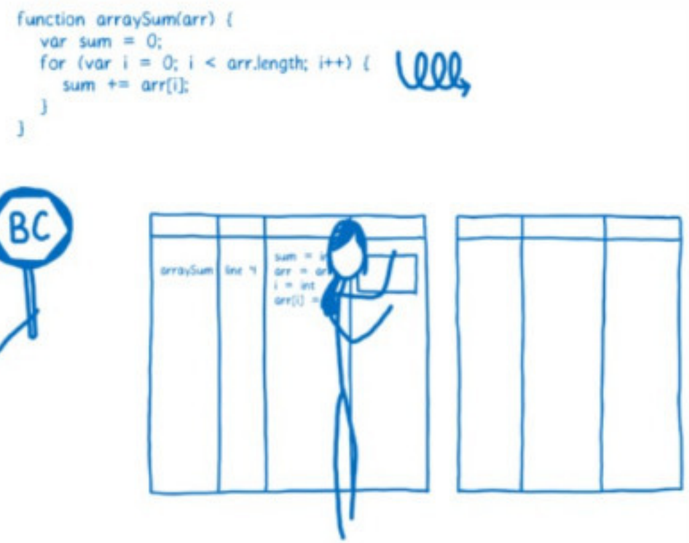








Just-in-time compilers

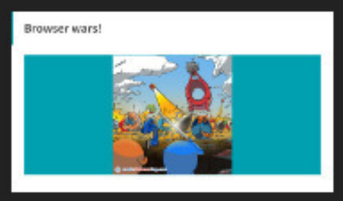


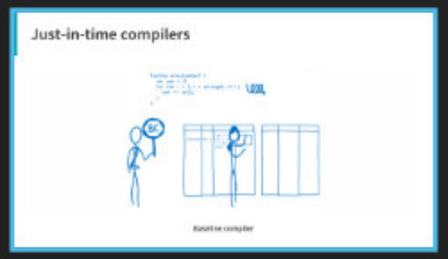
Baseline compiler

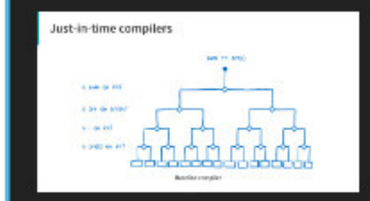
11:46:59 AM

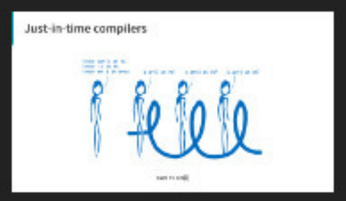
00:00:00

when the same line runs again and again, the monitor marks it as warm. and has it sent to the baseline compiler, then it saves a reference to the compiled code so whenever the line is hit again it just pulls the compiled version. it is worth to note again, compilation takes valuable processor time. but if a line runs over and over again, it might be worth it.

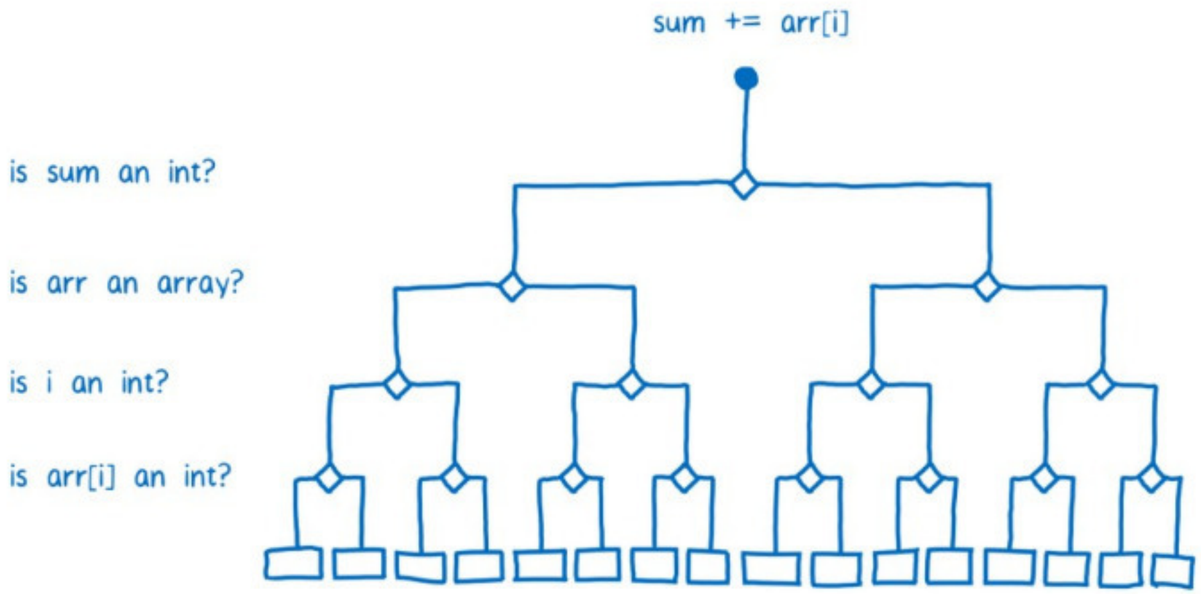









Just-in-time compilers

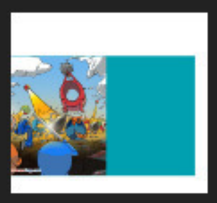


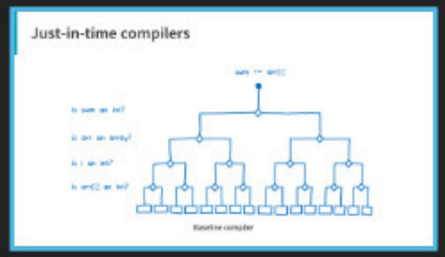
Baseline compiler

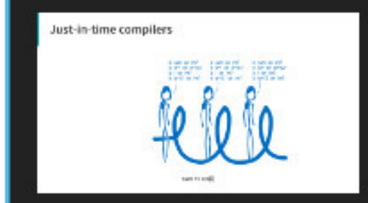
11:47:04 AM

00:00:00

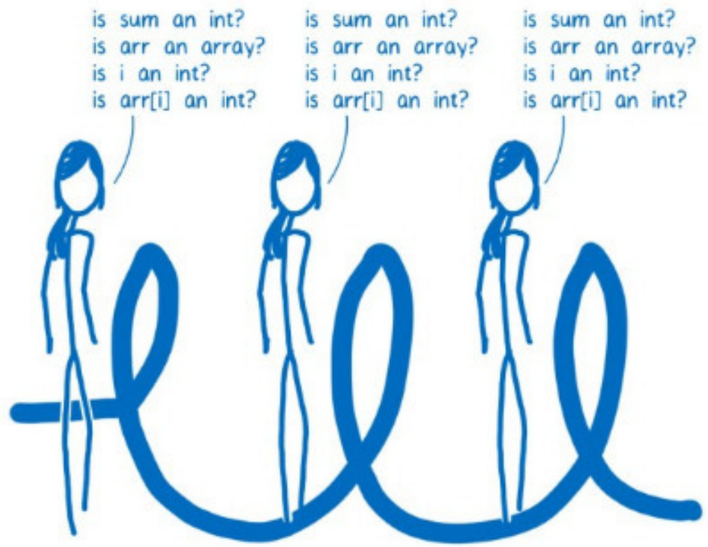
BUT, we're in JS, and that one line of code could mean different things for different variable types. string addition is a very different processor instruction than integer or floating point addition. in the example, if even one of the array's values has a different type then we need a compiled version to account for it. the baseline compiler is very much aware of this and creates different compiled versions for different inputs. but this means that before execution of the compiled code it still needs to ask itself what is the shape of the variables. in a loop this can be very expensive.





Just-in-time compilers



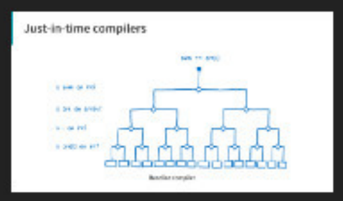
sum += arr[i]

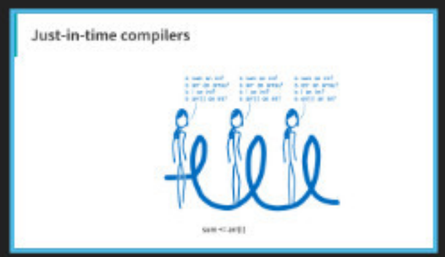
11:47:10 AM

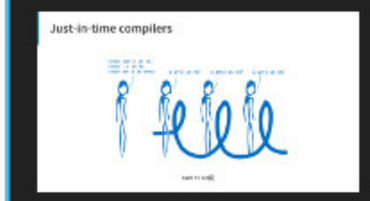
16/50

00:00:02

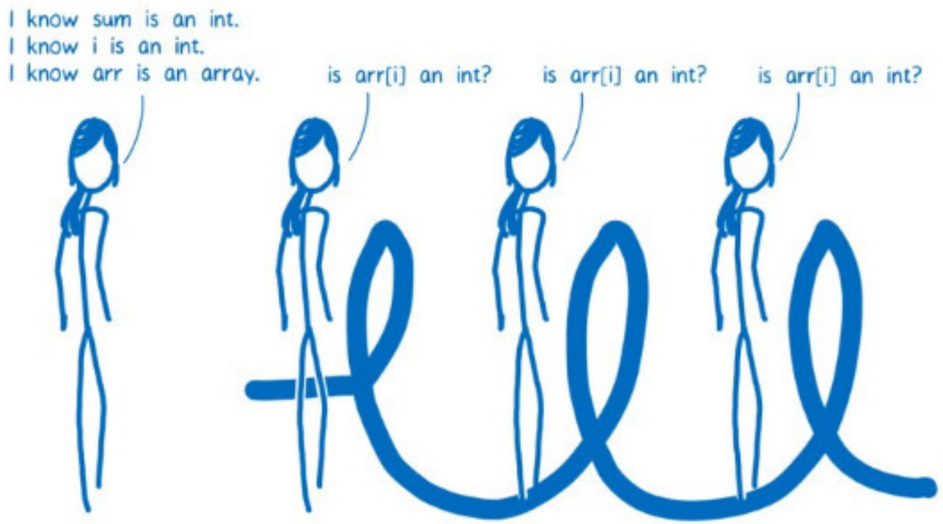
Then at runtime the profiler makes all these type assertions to choose the correct path to take. that's expensive!





Just-in-time compilers



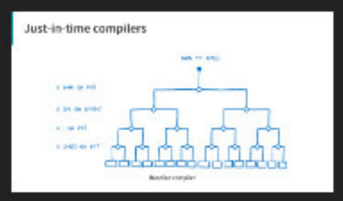
sum += arr[i]

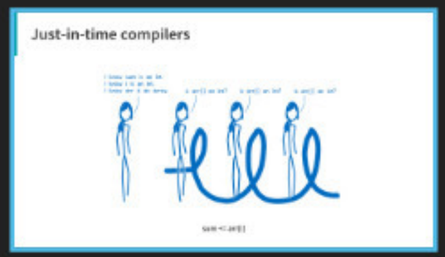
11:47:16 AM

17/50

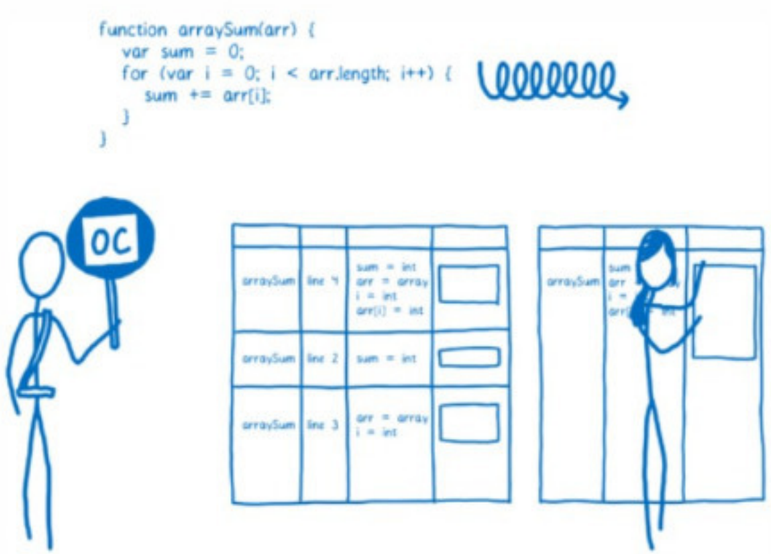
00:00:00

one optimisation we could do for this loop for example is move some of these assertions outside of the array. that's exactly where the optimising compiler comes in.



Just-in-time compilers - 10x faster!



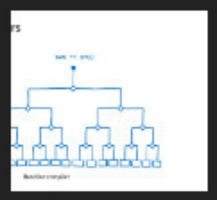
Optimizing compiler

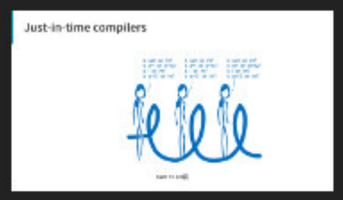
11:47:20 AM

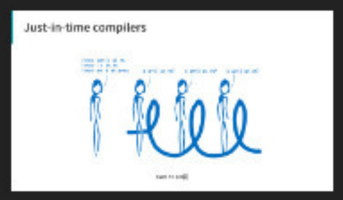
18/50

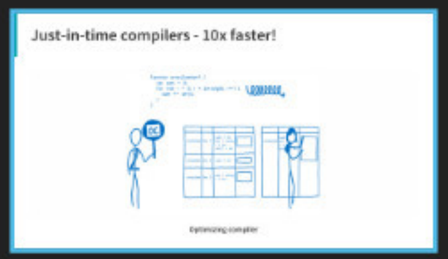
00:00:00

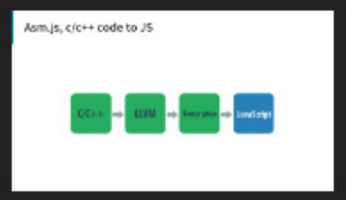
when a warm piece of code really runs and marked as hot it's really worth it to make all these optimisations ASSUMPTIONS OVER THE CODE! to provide with a very fast set of instructions that only run under those assumptions this optimisation step takes a long time so obviously it doesn't scale for every line of code. but if a line runs a lot then it's worth it. one caveat. if the assumptions turn out to be false, then we BAIL OUT. if they are right then we end up running code that's a lot faster than regular interpreter code. overall JIT compiler run approx 10x faster than reg. off course, this is only one optimisation step











Section 2

History of WebAssembly

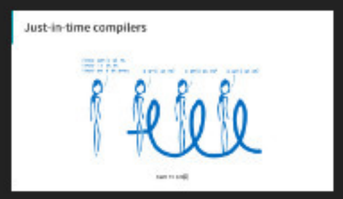
ASM.js

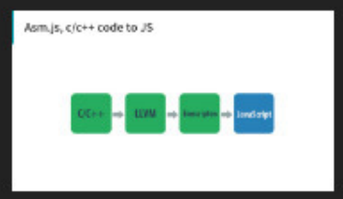
11:47:24 AM

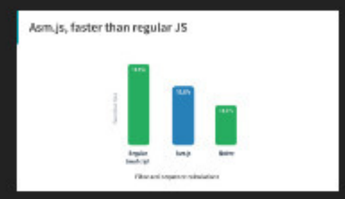
19/50

00:00:00

developed by mozila First appeared 21 March 2013; 5 years ago[1] asm.js is a subset of JavaScript designed to allow computer software written in languages such as C to be run as web applications while maintaining performance characteristics considerably better than standard JavaScript. the main idea is to use a version of JS thats strict enough to allow for easier compiler optimisations and fewer to no bailouts. one example of this is elimination of dynamic types.







https://www.beautiful.ai/presenter

Asm.js, a strict subset of JavaScript

```
var first = 5;
var second = first;
```

javascript

```
var first = 5;
// By using a bitwise operator,
// we make sure that the value is 32-bit integer
var second = first | 0;
```

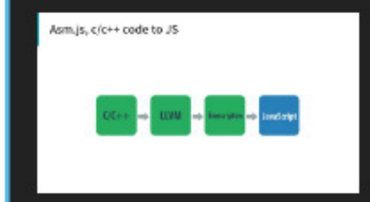
asm.js

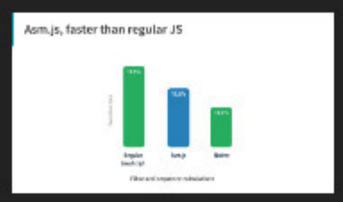
11:47:28 AM

20/50

00:00:00

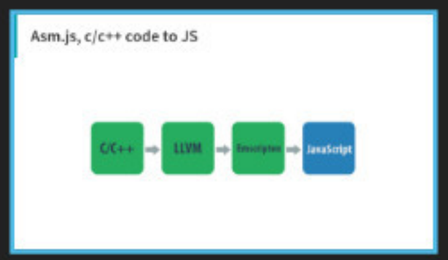
remember how hard it was for the optimising compiler to pick hot lines? By using the bitwise operator we convert the value of the first variable to a 32-bit integer. This ensures that second is always treated as a 32-bit integer. asm.js has a number of other similar rules. By combining these rules with normal JavaScript, much faster code can be created skipping a lot of optimisation steps . no one wants to write, let alone maintain code like this.

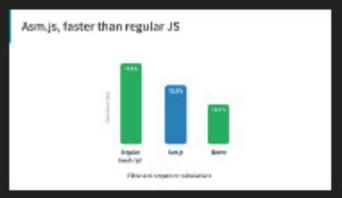




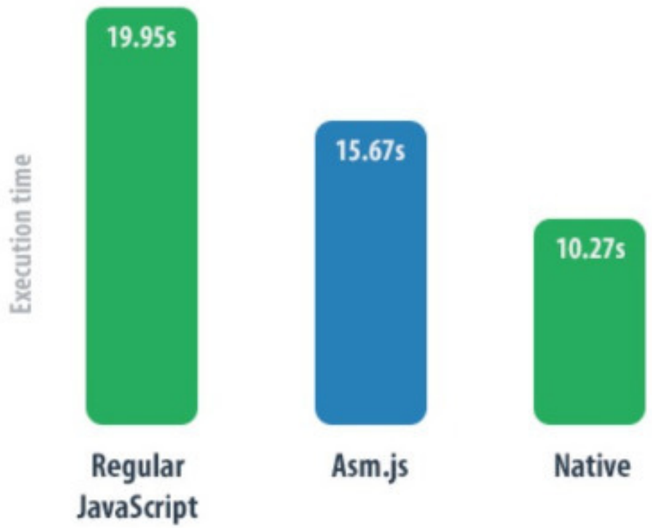
Asm.js, c/c++ code to JS

llvm - compiler infra. empscripten - takes llvm bitcode to js. it is not a good idea to write asm.js code by hand. The result would be hard to maintain and time consuming to debug. The good news is that a few tools exist for generating JavaScript code according to the asm.js specification from other languages like C or C++. EMSCRIPTEN is one of them, a mozila project.





Asm.js, faster than regular JS



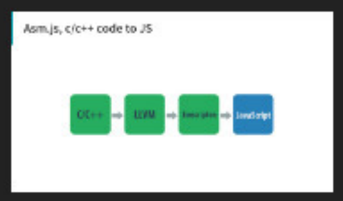
Fibonacci sequence calculations

11:47:36 AM

22/50

00:00:00

checking against emscripten compiled version of c code for Fibonacci sequence against regular js. you see around 25% faster. still slower than native code. in 2013, ported Unreal Engine 3 ported to Asm.js in four days - enabled for the first time. it's so straight forward to port these codebases to asm.js that some time ago we had this! show this: https://win95.ajf.me/win95.html



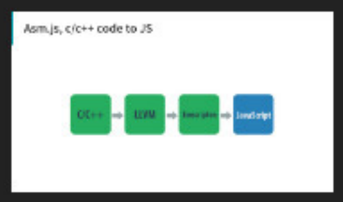
Section 3 - (finally)

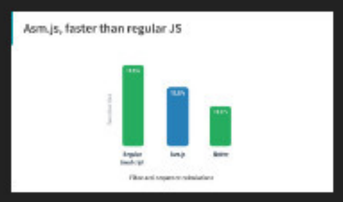
WebAssembly

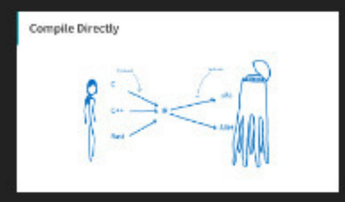
11:47:40 AM

00:00:00

Finally, after going through the history of how JS runs in our browser, and the uprising of asm.js trying to get code run even faster. we move on to the next iteration. I'de like to invide Oded Sofrin of Wix Photographers to tell us more.







WebAssembly.org

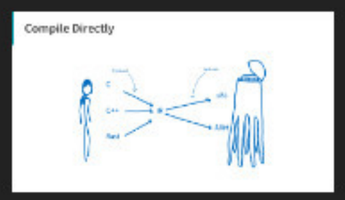
WebAssembly Working Group.

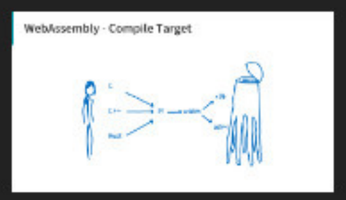
11:47:44 AM

24/50

00:00:00

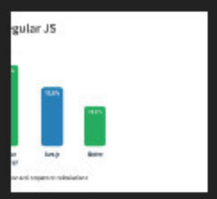
Since 2015 Working Group: Google Mozlla, apple, facebook, intel, lg, microsoft No more optimzing and adding layers to that interpreter from 95, but starting from scratch an idea that fits the current usage of browsers 2017 - WASM MVP WASM is binary instruction format for stack-based virtual machine portable target for compilation of languages like C/C++/Rust (and many more to come) enable deployment on the web for client/server apps

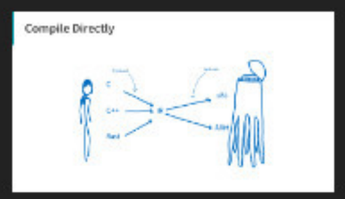


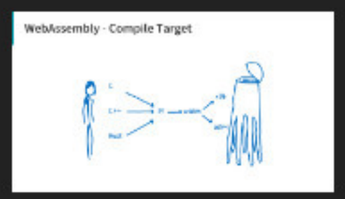


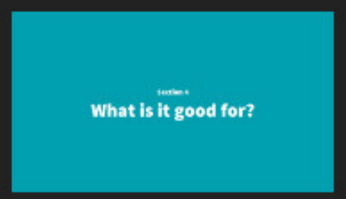
```
(func (param $p i32)
  get_local $p
  get_local $p
  i32.add)
```


it is a stack based binary instruction that the browser can run in near native performance. efficency - binary size & decode / compilation wise compiled target (nobody wants to write like that)

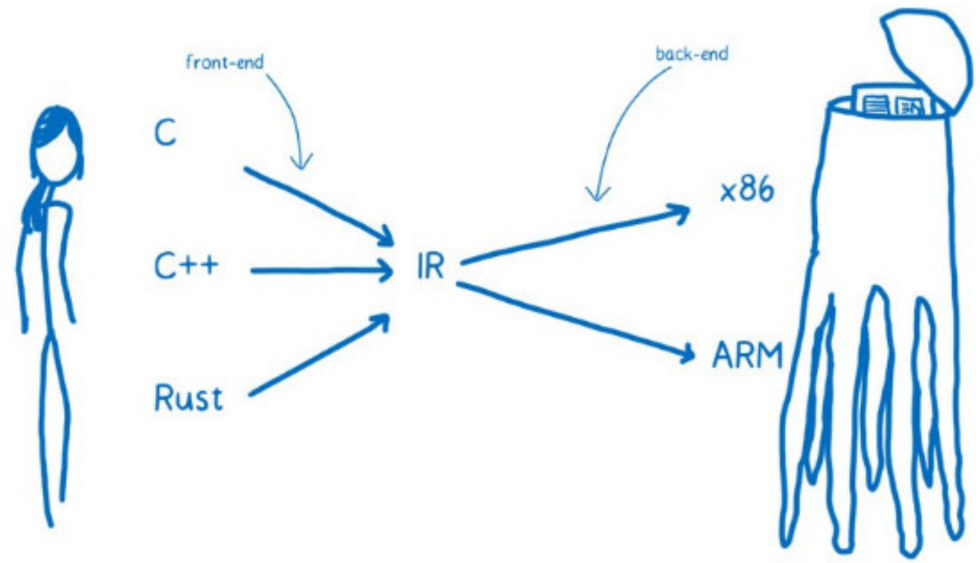








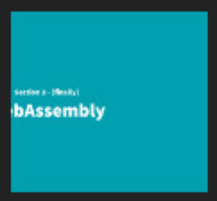
Compile Directly



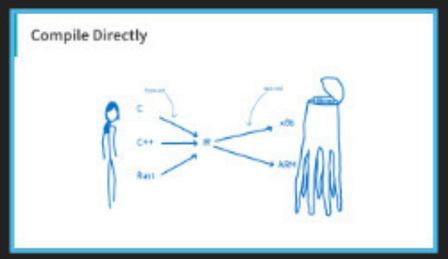
11:47:53 AM

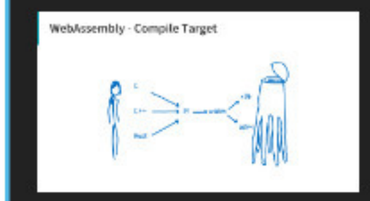
00:00:00

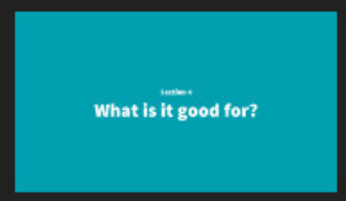
compiling directly: compile corresponds to a particular machine architecture (windows / mac / android...) INTERMIDIATE REPRESENTATION When you're delivering code to be executed on the user's machine across the web, you don't know what your target architecture the code will be running on. You might think WebAssembly as just another one of the target assembly languages. WebAssembly is a little bit different than other kinds of assembly. It's a machine language for a conceptual machine, not an actual, physical machine. It is not primarily intended to be written by hand, rather it is designed to be an effective compilation target for low-level source languages like C, C++, Rust, etc.

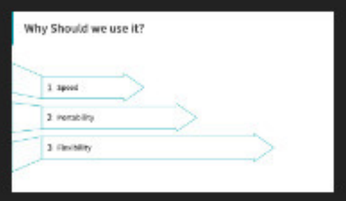




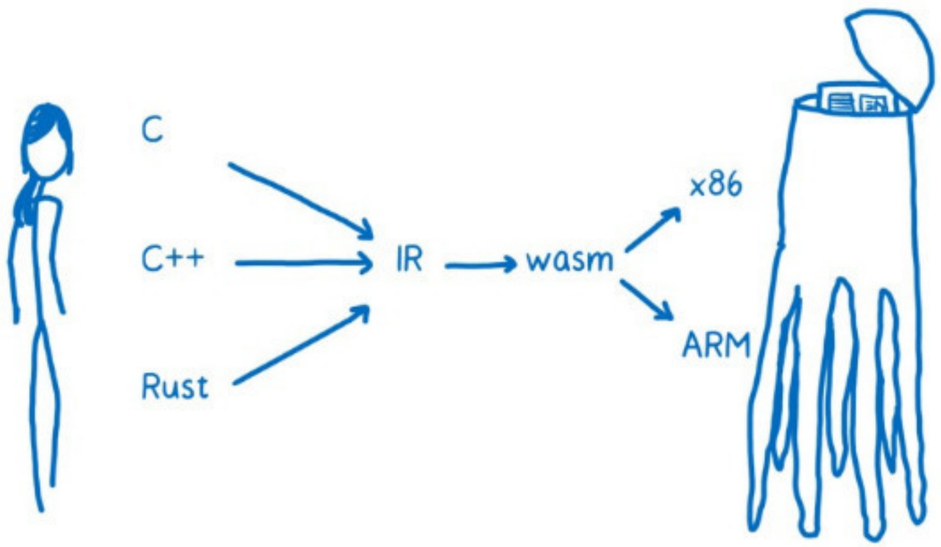








WebAssembly - Compile Target

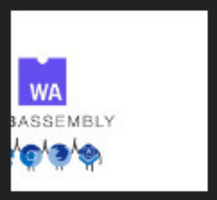


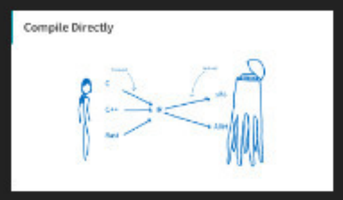
11:47:58 AM

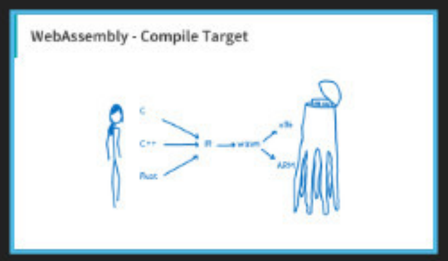
27/50

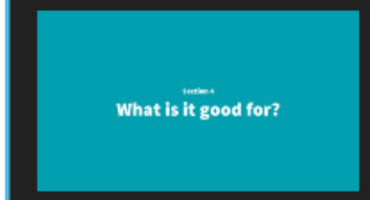
00:00:01

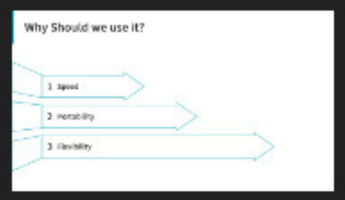
Because of this, WebAssembly instructions are sometimes called virtual instructions. They have a much more direct mapping to machine code than JavaScript source code. but they are not assembly. They represent a sort of intersection of what can be done efficiently across common popular hardware. But they aren't direct mappings to the particular machine code of one specific hardware. The browser downloads the WebAssembly. Then, it can make the short hop from WebAssembly to that target machine's assembly code.

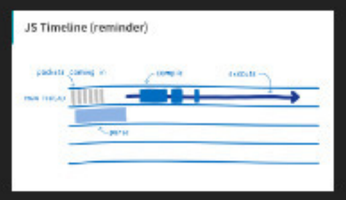








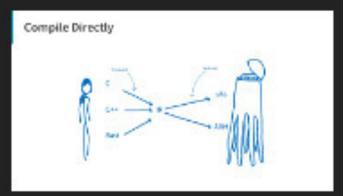


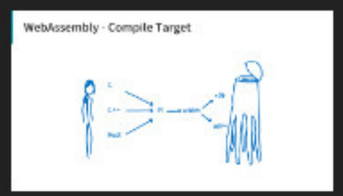


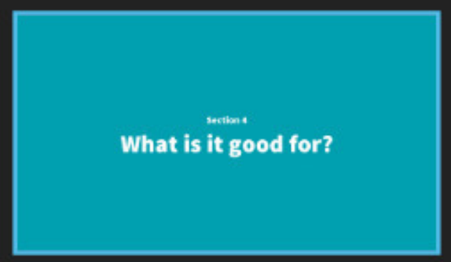
Section 4

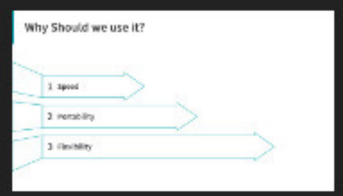
What is it good for?

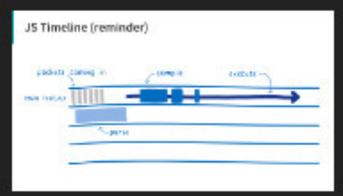
No notes

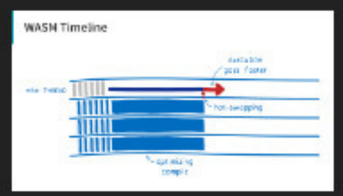












Why Should we use it?

1 Speed

2 Portability

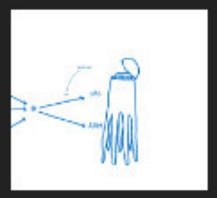
3 Flexibility

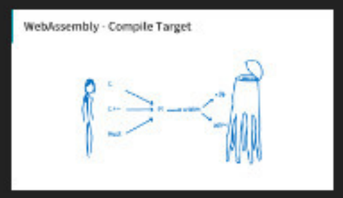
11:48:06 AM

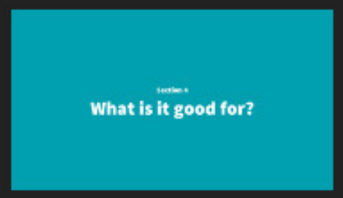
29/50

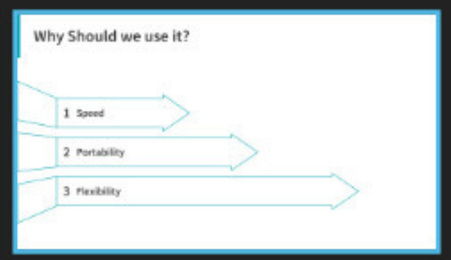
00:00:00

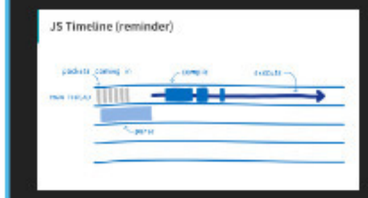
No notes

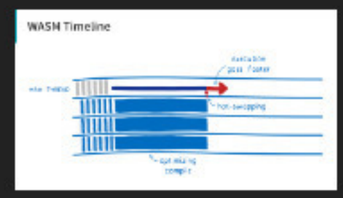


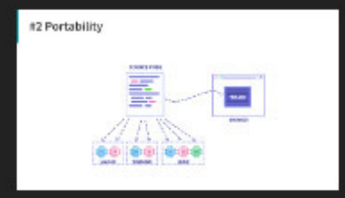




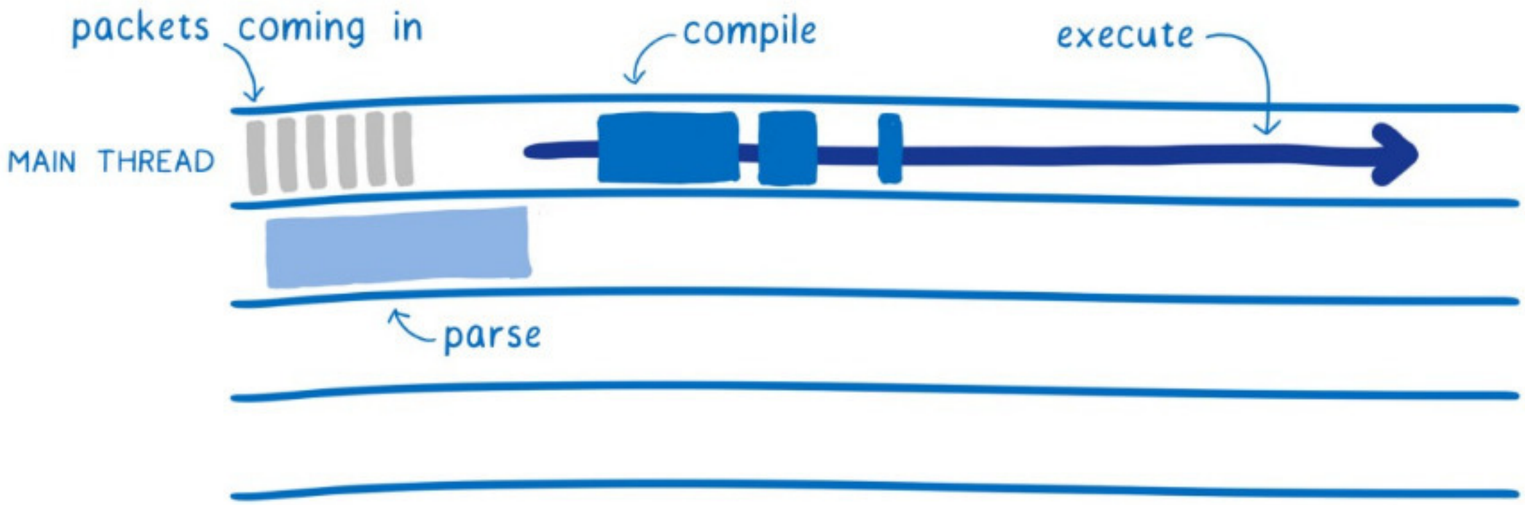








JS Timeline (reminder)

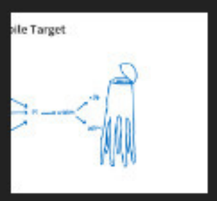


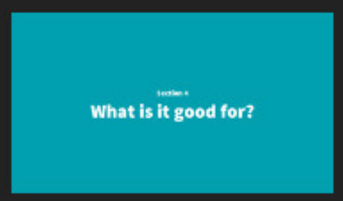
11:48:11 AM

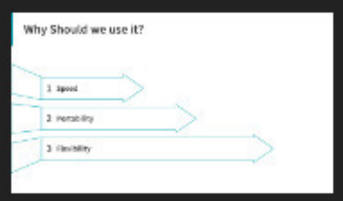
30/50

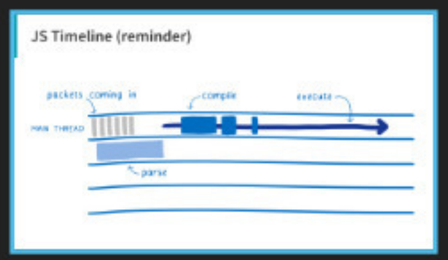
00:00:01

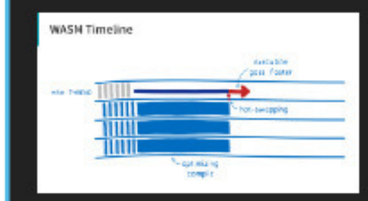
Speed Even in the optimal mode, when we start PARSING as the packets comes in, on a different thread.. we can only start executing when we done parsing, on the main thread, interrupted by optimization & re-optizimations

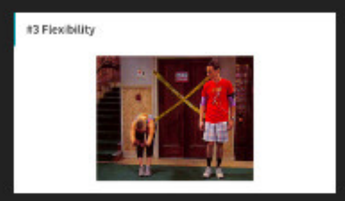




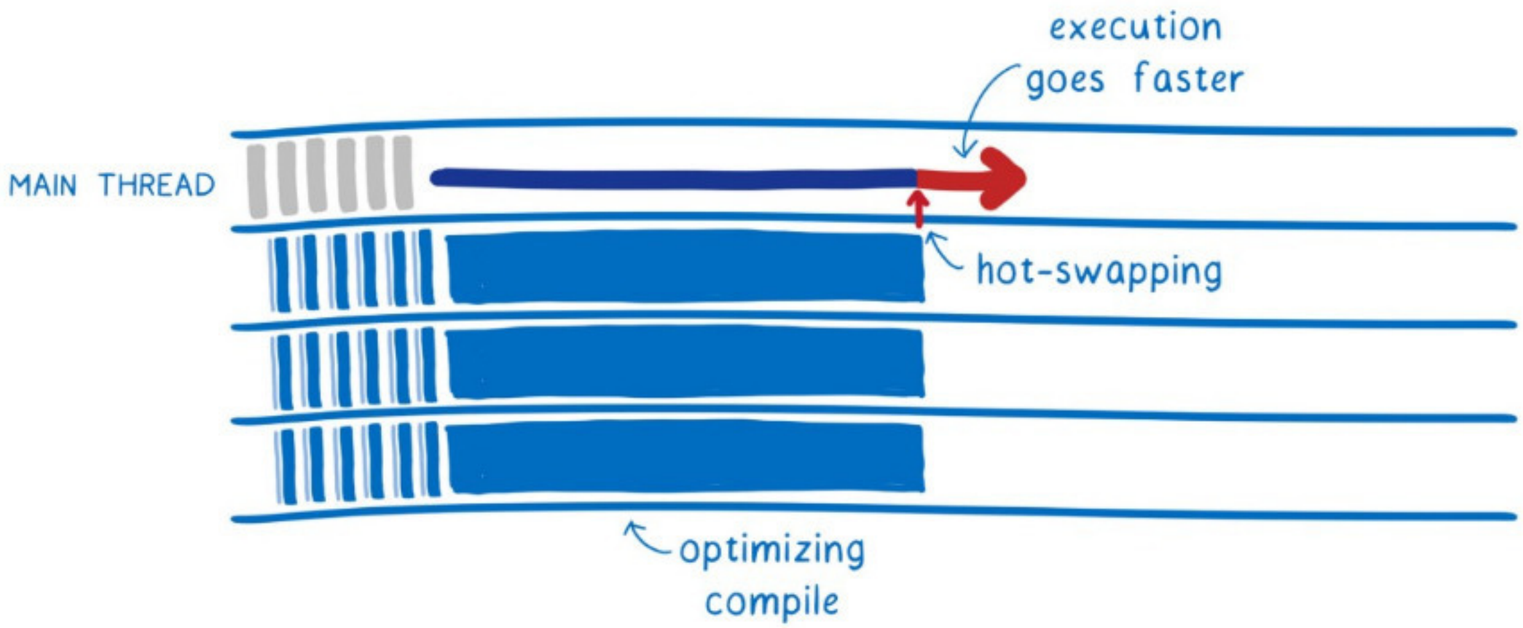








WASM Timeline

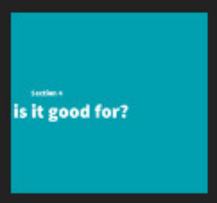


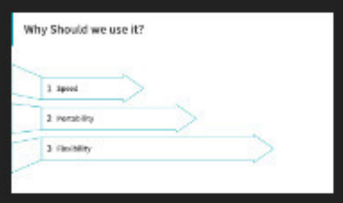
11:48:15 AM

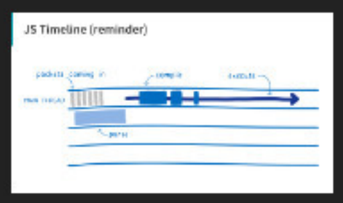
31/50

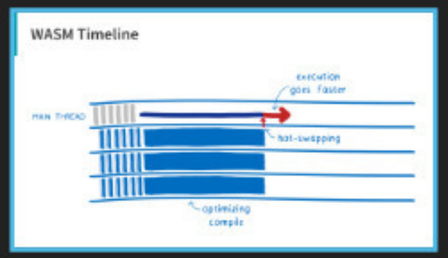
00:00:00

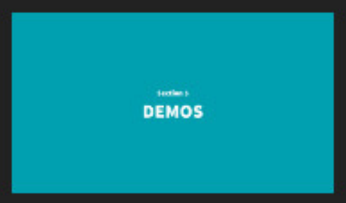
Decode as oppose to parsing Compiling is much faster - A lot of the optimzation were already done in the transition to wasm - written in ideal language for machines - types are well defined, so one compilation fits all Different browsers handle compiling WebAssembly differently. Some browsers do a baseline compilation of WebAssembly before starting to execute it, and others use a JIT. In current FF - packets are being decoded & compiled (baseline) as packets come in, meaning it's takes not much longer than downloading, and start executing right away meanwhile on different multi-threads, optimization is being perform and hot-swapping on the main thread when done Download - Because WebAssembly is more compact than JavaScript, fetching it is faster. Even though compaction algorithms can significantly reduce the size of a JavaScript bundle, the compressed binary representation of WebAssembly is still smaller.



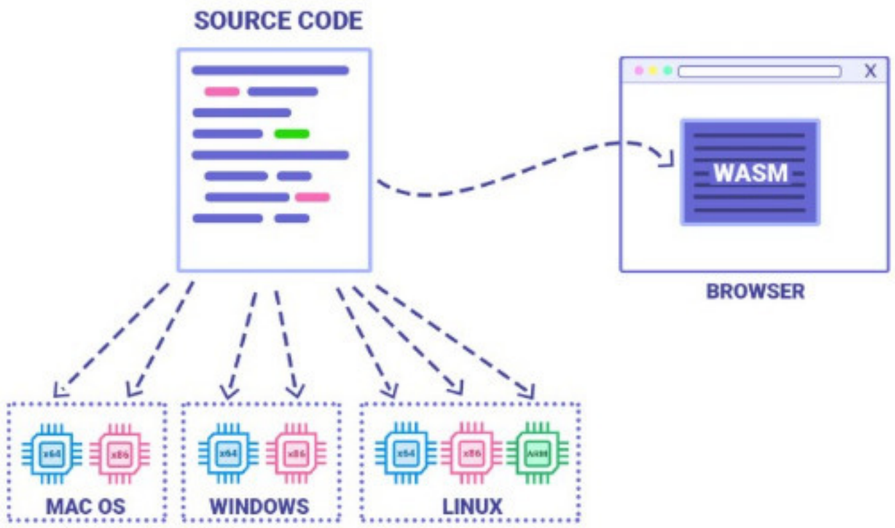








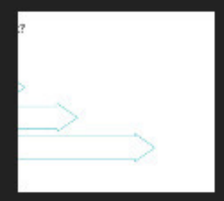
#2 Portability

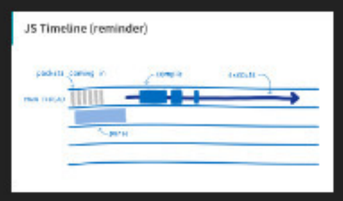


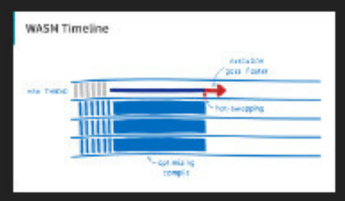
11:48:20 AM

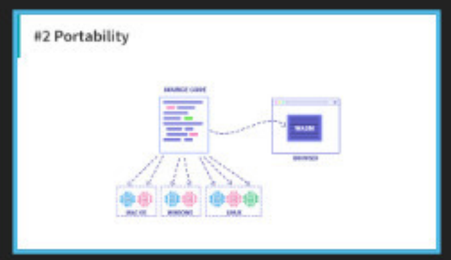
00:00:00

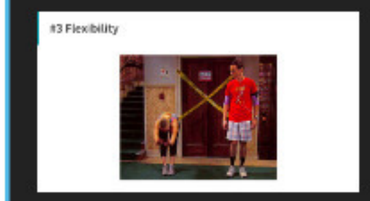
To run an application on a device, it has to be compatible with the device's processor architecture and operating system. That means compiling source code for every combination of operating system and CPU architecture that you want to support. With WebAssembly there is only one compilation step and your app will run in every modern browser (including mobile devices) (incredible wealth of C++ libraries and open source applications that exist out there)

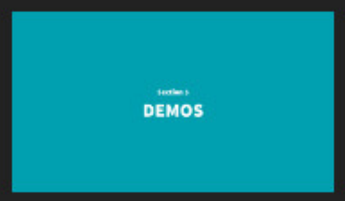


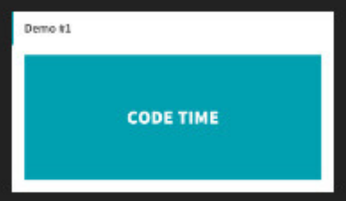


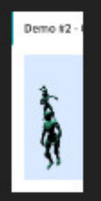






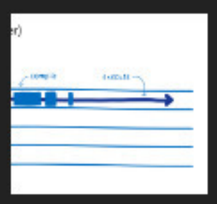


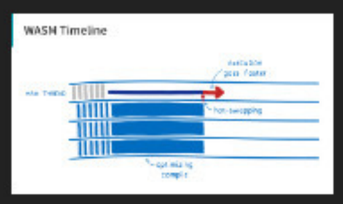


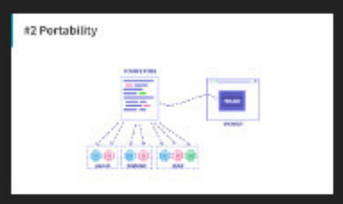


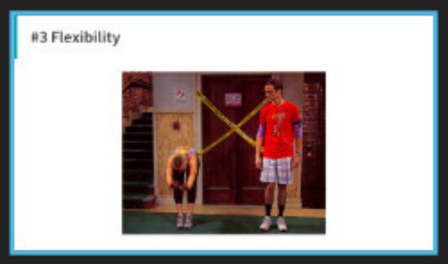
#3 Flexibility

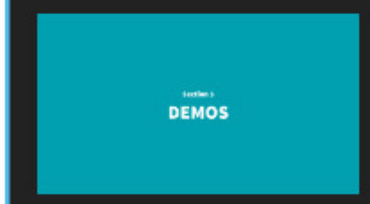
Web developers will be able to choose other languages and more developers will be able to write code for the web. JavaScript will still be the best choice for most use cases but now there will be an option to drop down to a specialized language once in a while when you really need a boost. Parts like UI and app logic could be in JavaScript, with the core functionality in WebAssembly. When optimizing performance in existing JS apps, bottlenecks could be rewritten in a language that is better suited for the problem.

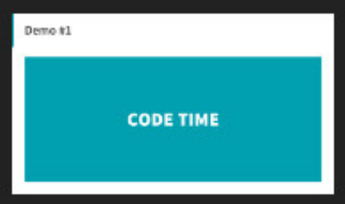


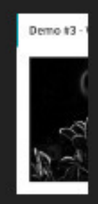












Section 5

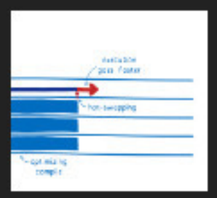
DEMOS

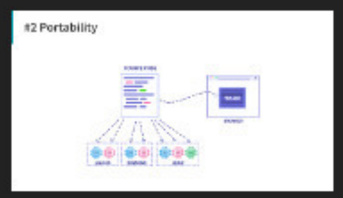
No notes

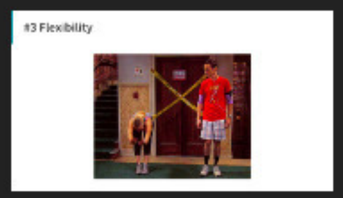
11:48:30 AM

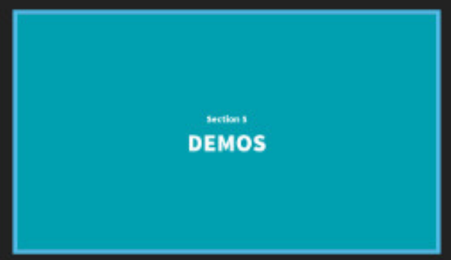
34/50

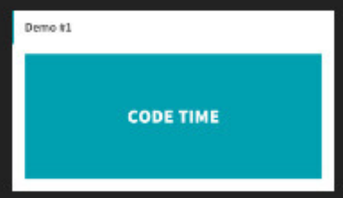
00:00:01

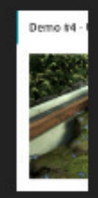








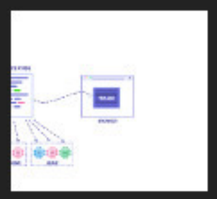


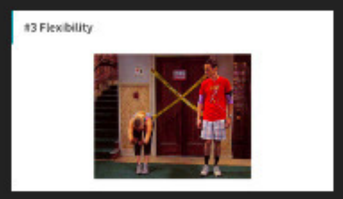


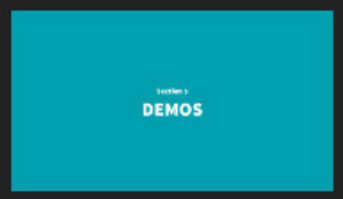
Demo #1

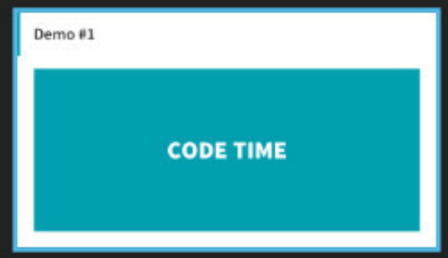
CODE TIME

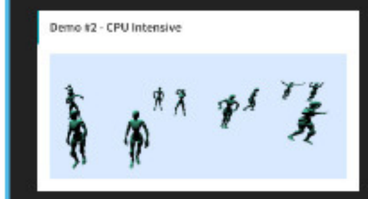
No notes

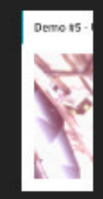




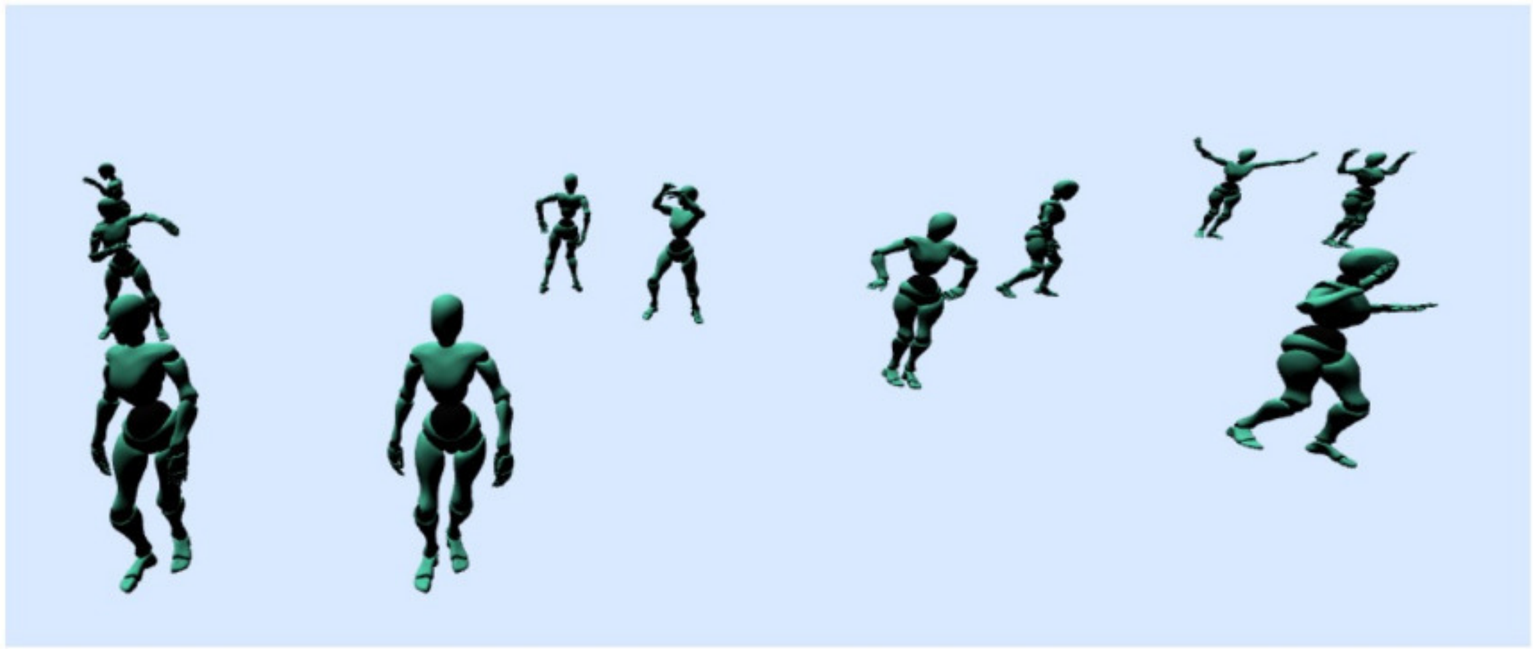




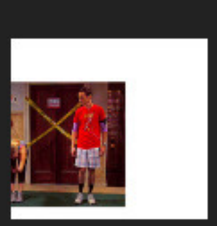


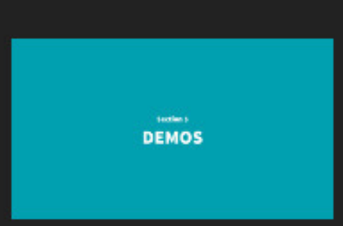


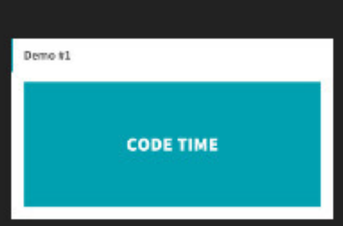
Demo #2 - CPU Intensive

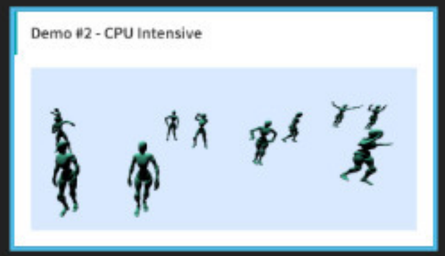


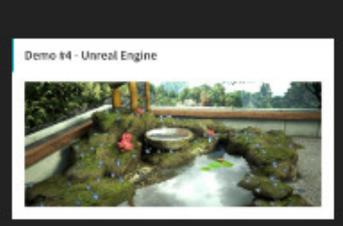
http://aws-website-webassemblyskeletalanimatffaza.s3-website-us-east-1.amazonaws.com/
skeletal animation system, animates multiple instances of a character across the screen almost identical implemenation in C++ and JS WebGL is used to power the 3D graphics

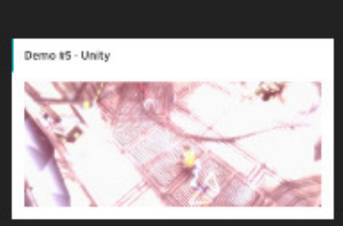






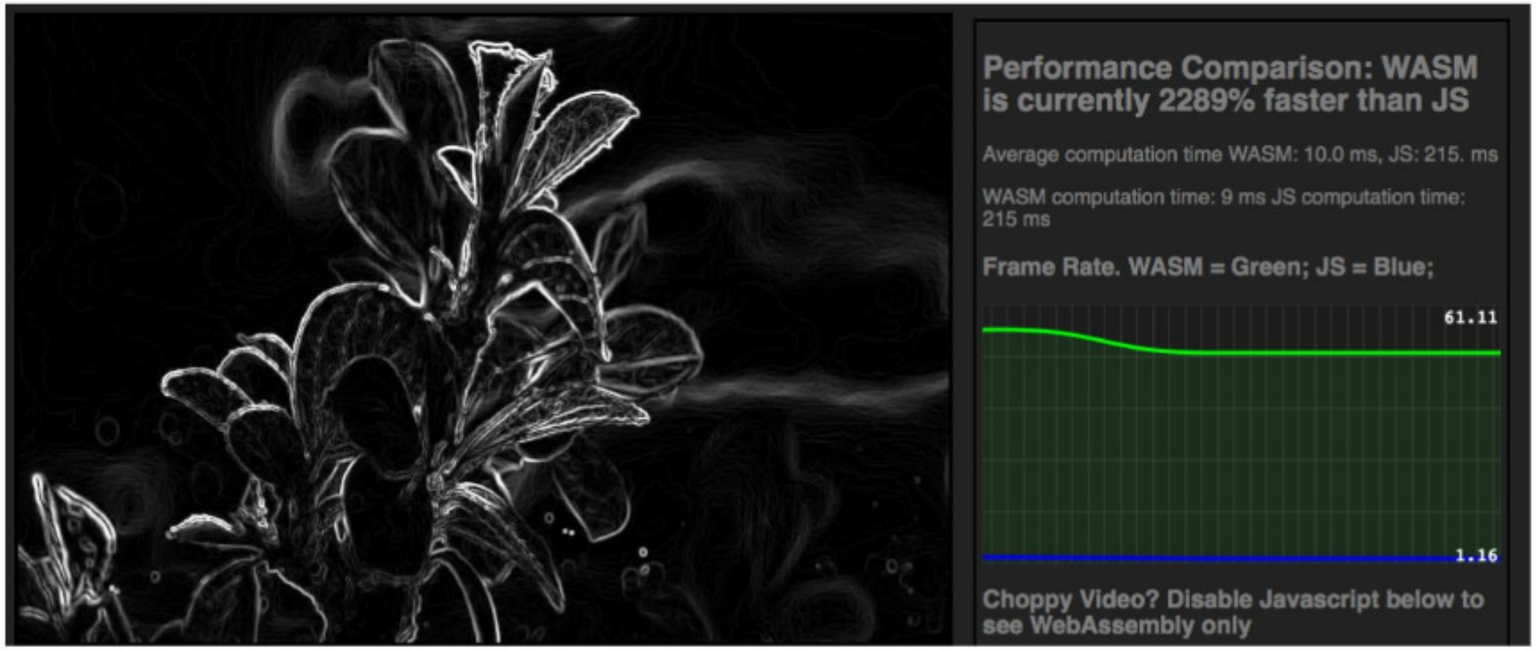






https://www.beautiful.ai/presenter

Demo #3 - Video Editor



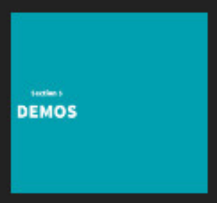
11:48:47 AM

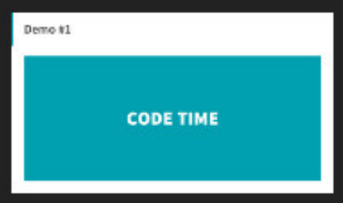
37/50

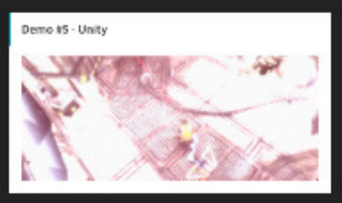
00:00:00

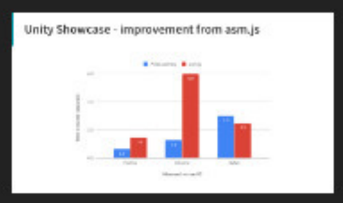
https://d2jta7o2ze

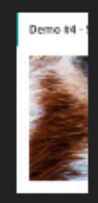
Not a silver bullet different behaviors between chrome & mozilla





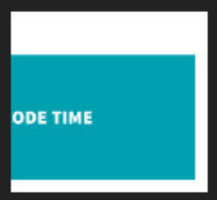


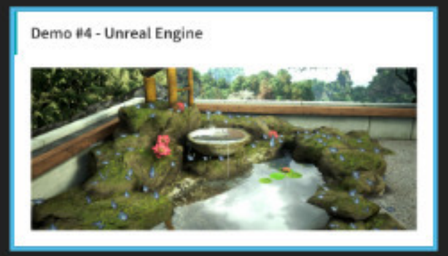


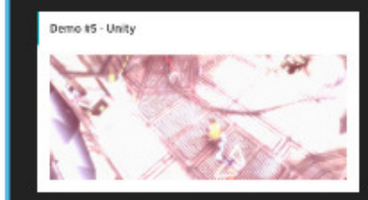


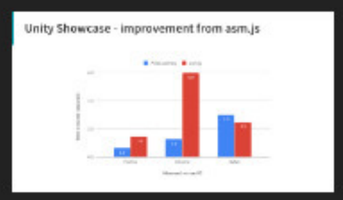
Demo #4 - Unreal Engine

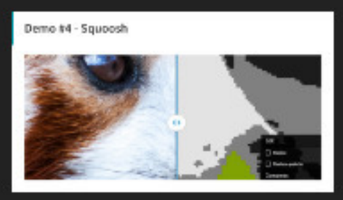
https://s3.amaz
games/ZenGard
Unreal and
unity are 3d
game engines





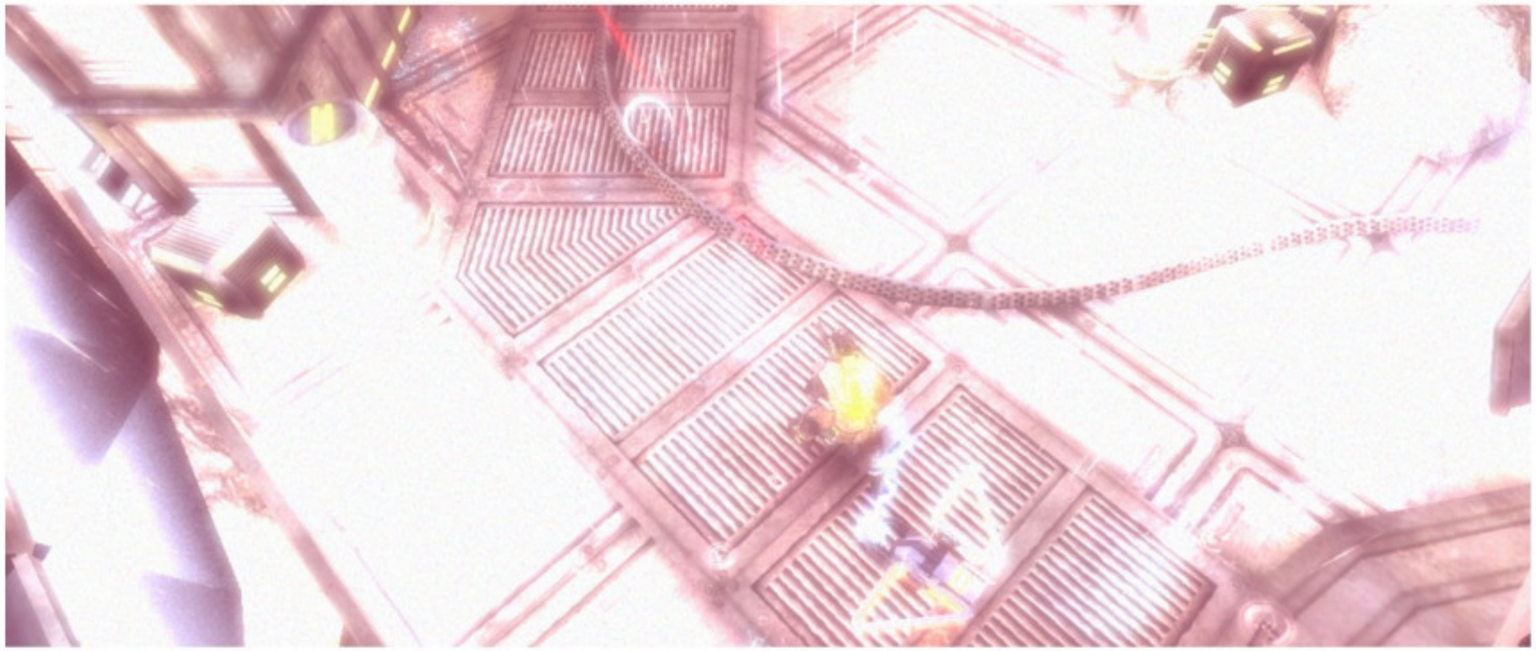






https://www.beautiful.ai/presenter

Demo #5 - Unity

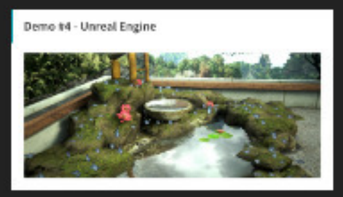


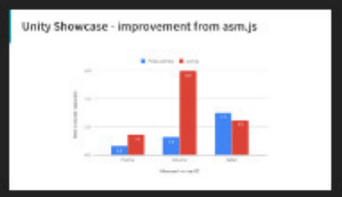
https://files.un

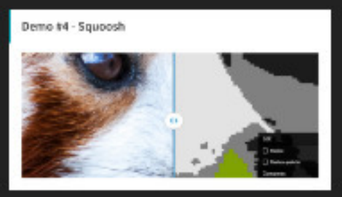
11:49:01 AM

39/50

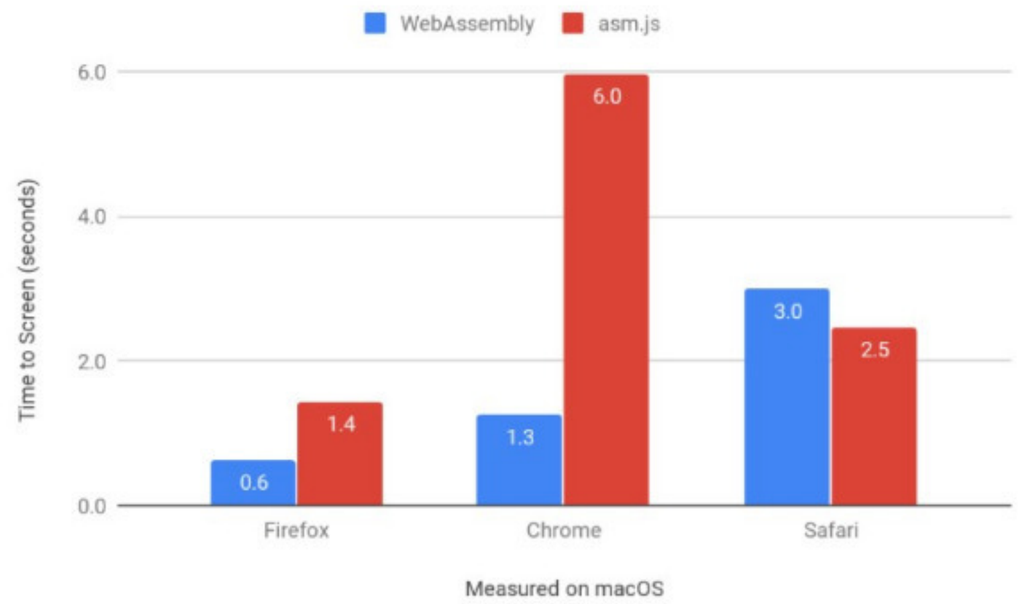
00:00:01







Unity Showcase - improvement from asm.js

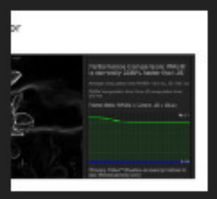


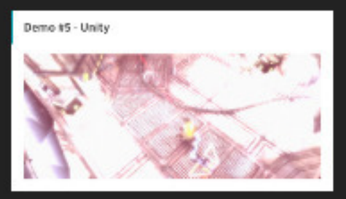
11:49:06 AM

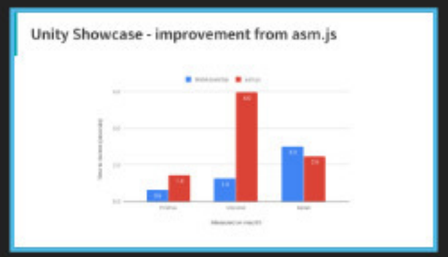
40/50

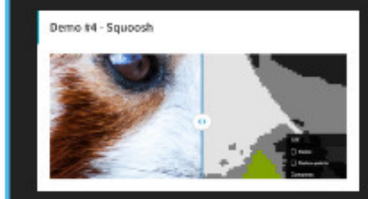
00:00:01

Unity also moved to wasm (august 2018) from asm.js - benchmarks asm vs wasm



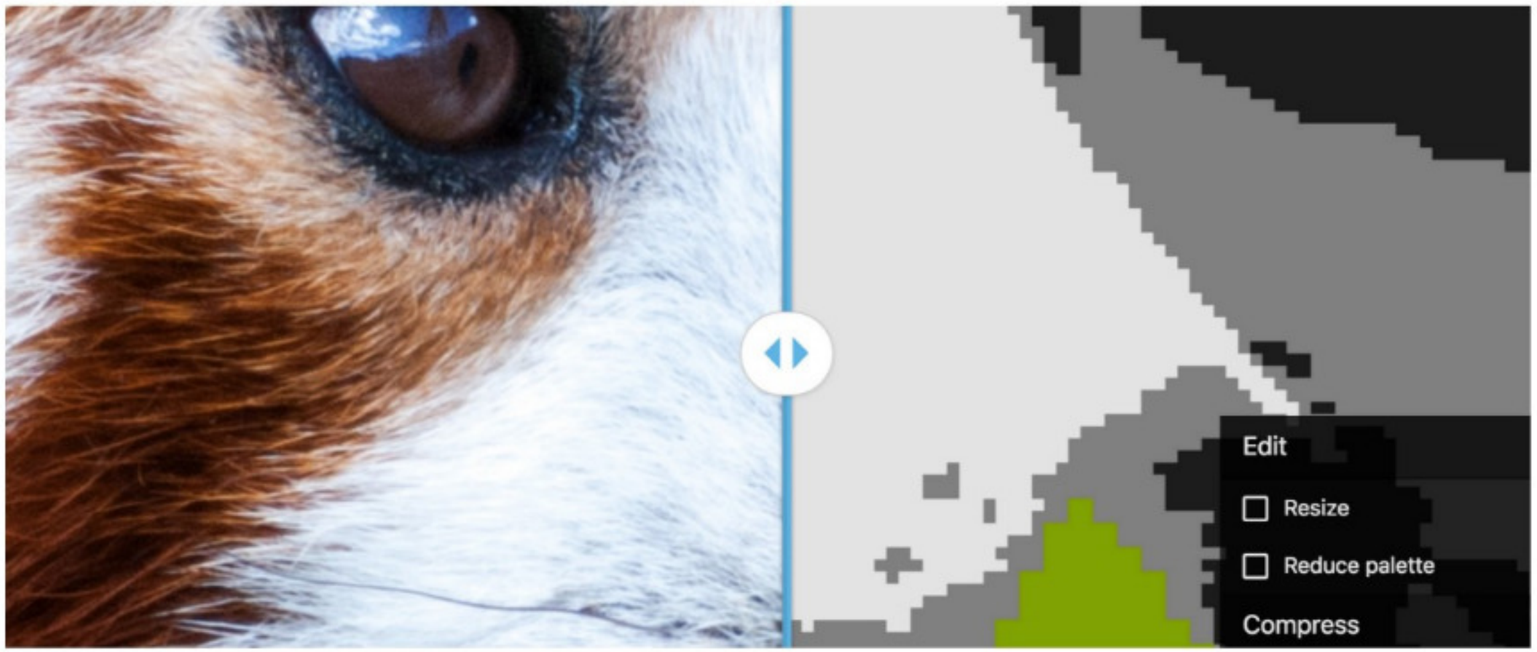








https://www.beautiful.ai/presenter

Demo #4 - Squoosh

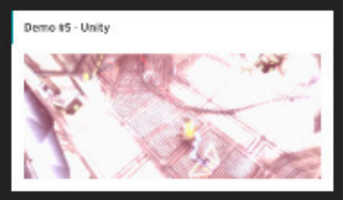


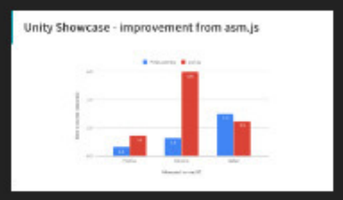
https://squoosh.a
from
Wedensday
showcase for
compression
mechanism
https://github.cor

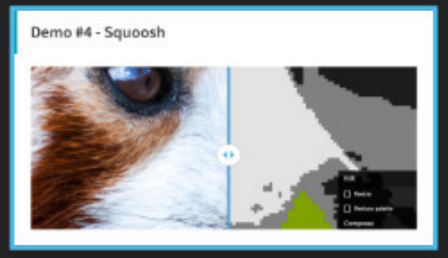
11:49:13 AM

41/50

00:00:02







November Holiday Hogan @AlanHogan · Aug 22

Replying to @WasmWeekly @rauschma

I just booted windows… on my phone… in a web browser… embedded in a frigging twitter client

1 5 19

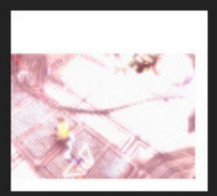
1 more reply

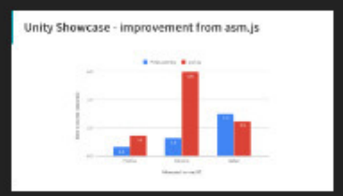
No notes

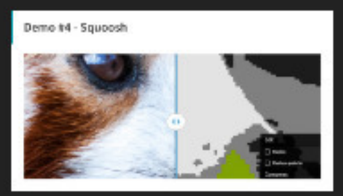
11:49:17 AM

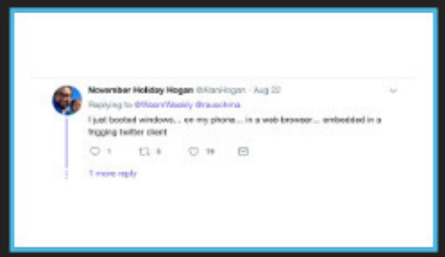
42/50

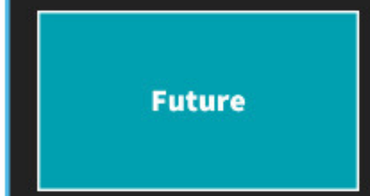
00:00:00











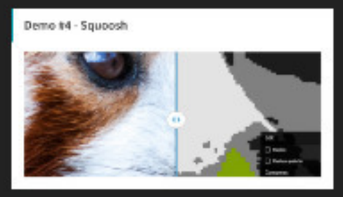
Future

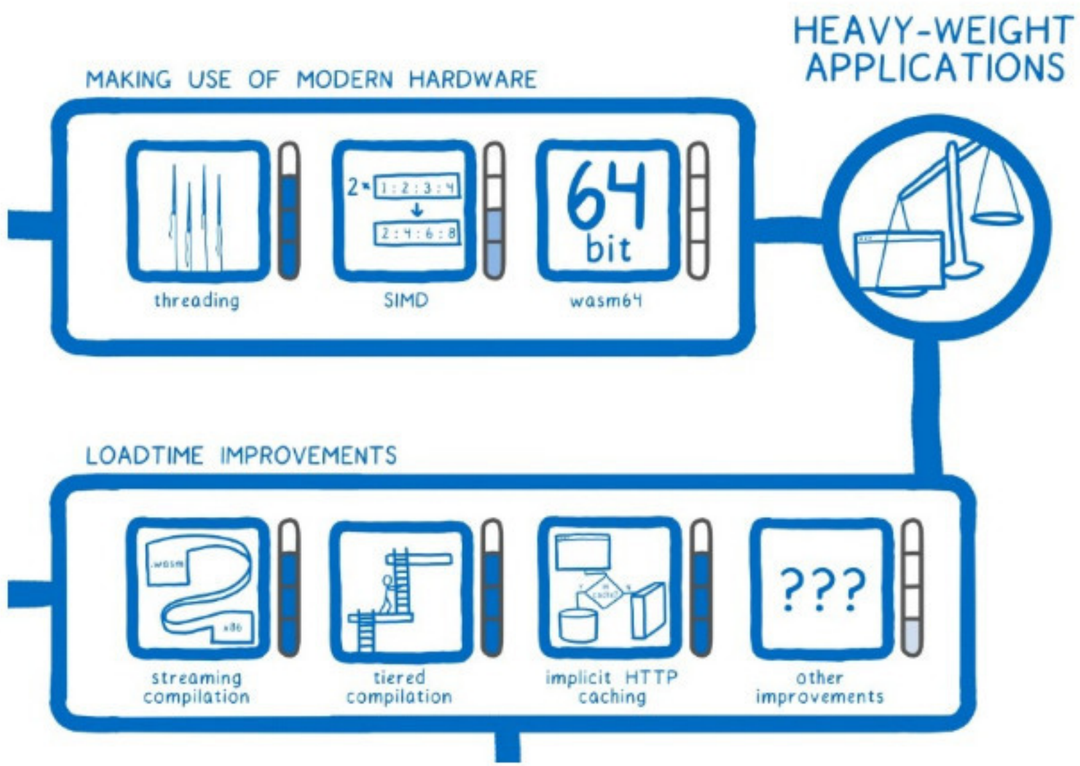
11:49:22 AM

43/50

00:00:00

pink world currrently it's an MVP let's talk about post-mvp world





11:49:27 AM

00:00:00

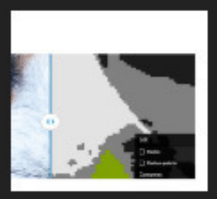
We already see some examples - Lightroom, AutoCAD software

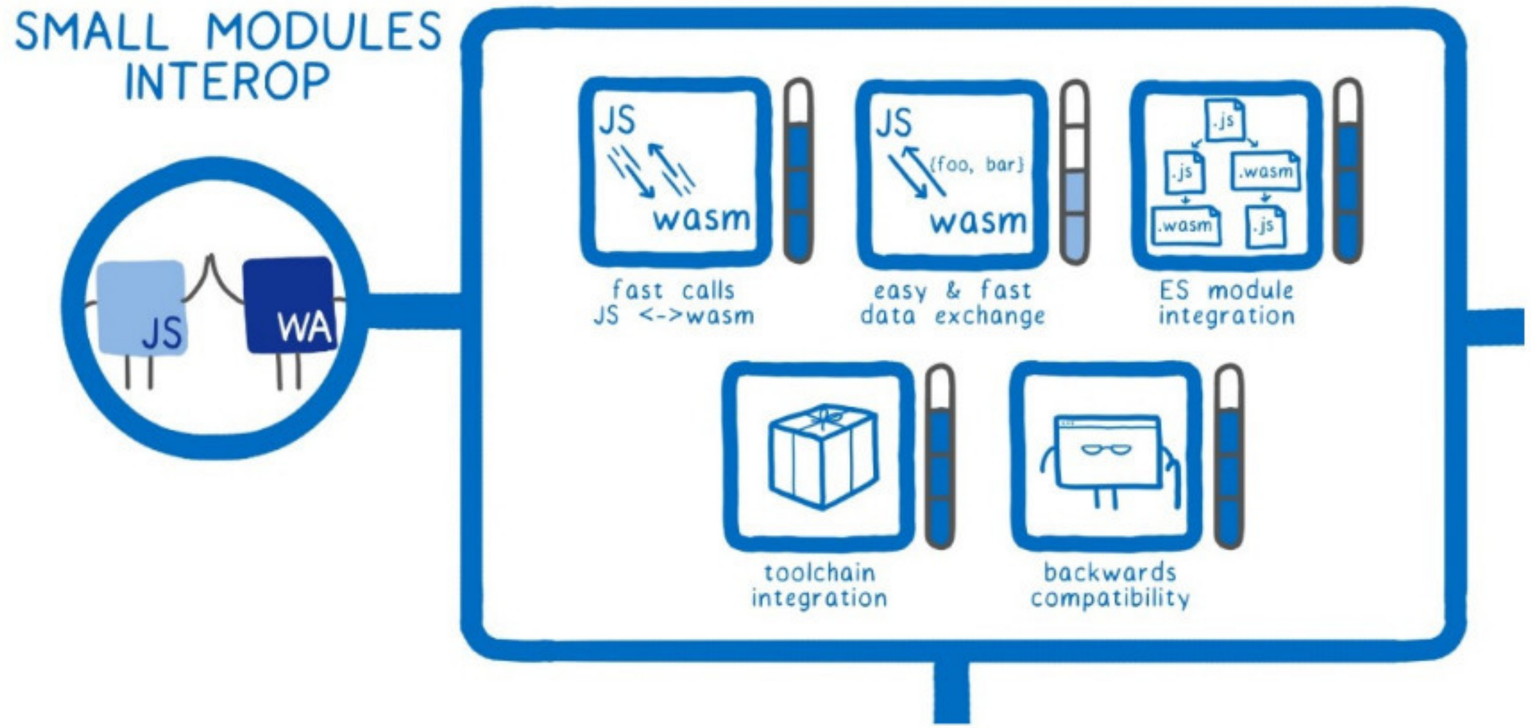
Multithreading support

SIMD - single instruction multiple data - parallel vector work

64 bit support

caching - store compiled code



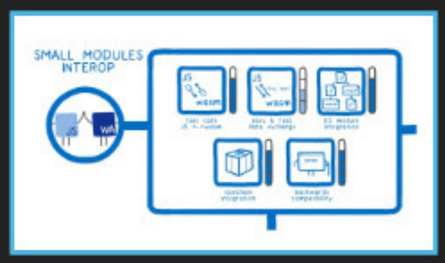


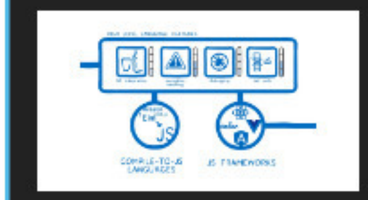
Easily use small models and switch between js packages to wasm pacakges WEB API es module integration - using import and export toolchain - npm for webassembly

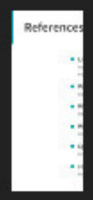
11:49:33 AM

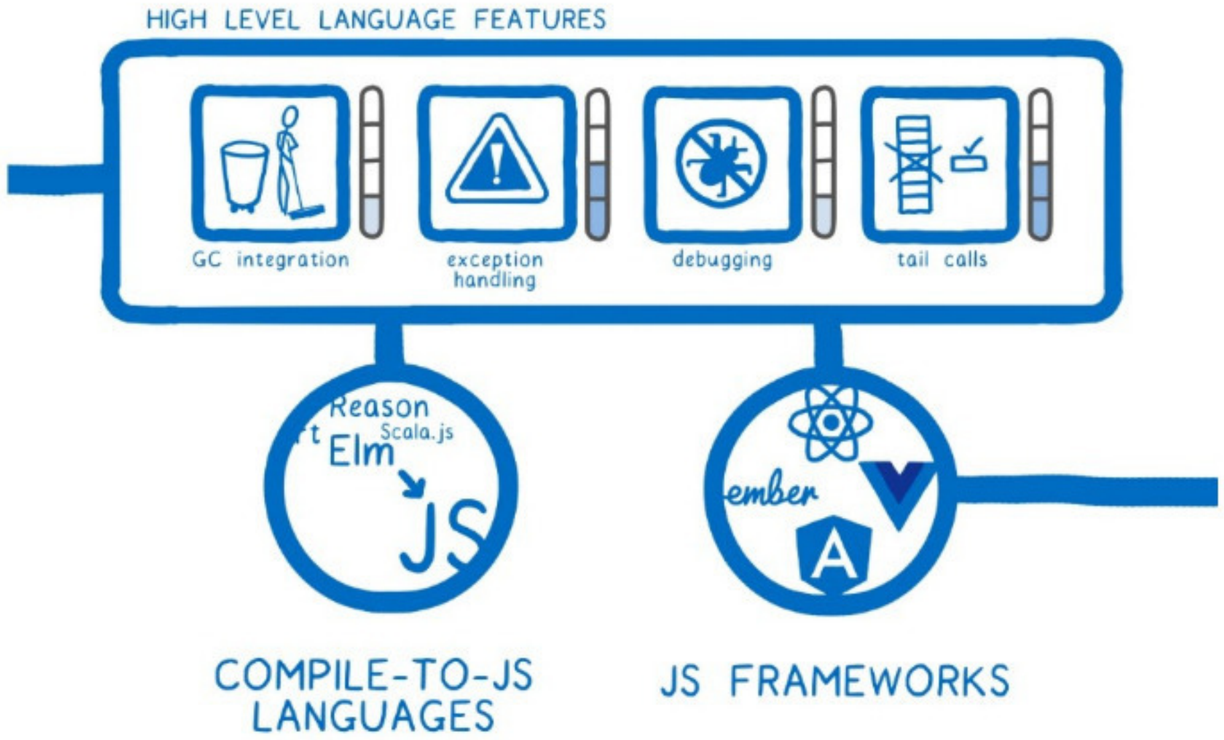
45/50

00:00:01







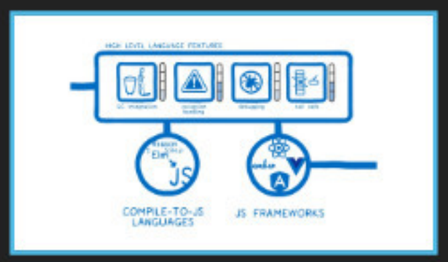


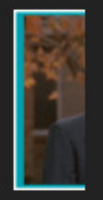
WASM ff example Frameworks - for example virtual dom diff for example compile to js - change the target to wasm instead of JS GC - cross-platform cycles, and performance tail call - performance

11:49:37 AM

46/50

00:00:00



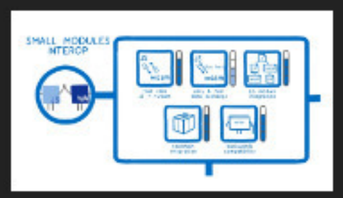


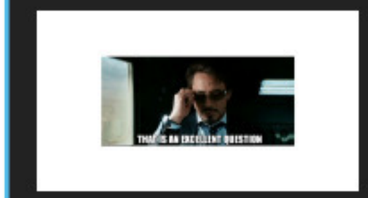
Questions?

11:49:41 AM

00:00:00

No notes



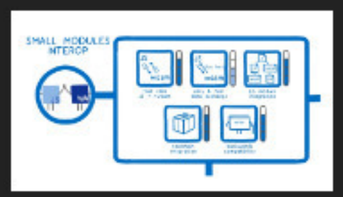


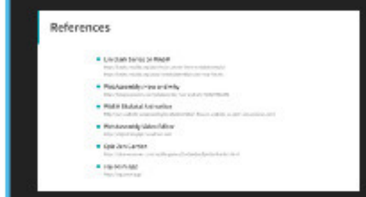
11:49:45 AM

48/50

00:00:00

No notes





References

- **Lin Clark Series on WASM**
  https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/
  https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/
- **WebAssembly: How and why**
  https://blog.logrocket.com/webassembly-how-and-why-559b7f96cd71
- **WASM Skelatal Animation**
  http://aws-website-webassemblyskeletalanimation-ffaza.s3-website-us-east-1.amazonaws.com/
- **WebAssembly Video Editor**
  https://d2jta7o2zej4pf.cloudfront.net/
- **Epic Zen Garden**
  https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html
- **squoosh.app**
  https://squoosh.app/

11:49:54 AM

00:00:01

No notes

11:50:12 AM

50/50

00:00:00

No notes