Frontend-Spielereien mit WebAssembly

Was hat ein Omni-Channel Buchhändler mit WebAssembly zu tun? Erstmal nicht viel. Trotzdem beschäftigen wir uns auch mit neuen, interessanten Technologien, welche nicht direkt in unserem täglichen (Entwickler) Leben anwendbar sind. Dieser Blog-Eintrag beschäftigt sich mit der Entstehung und dem aktuellen Stand der WebAssembly Spezifikation.

Die Gegenwart

Wir schreiben das Jahr 2018. Das Internet ist allgegenwärtig. Als Anwender und Entwickler haben wir eine Vielzahl an unterschiedlichsten Technologien für die Entwicklung moderner Web-Anwendungen zur Verfügung. Wir haben den HTML5 Standard. Wir haben CSS. Wir haben IndexedDB oder auch den LocalStore im Browser. Mit diesen Technologien lassen sich schnell und effizient Online- oder auch Offlineanwendungen bauen und verteilen. Seit Anfang 2017 gibt es neu WebAssembly in diesem Universum. Um WebAssembly und dessen Bedeutung genauer zu verstehen, müssen wir jedoch eine kleine Zeitreise in die Vergangenheit unternehmen.

Wie alles begann

Wir schreiben das Jahr 1995. In diesem Jahr wurde ein wichtiger Meilenstein für das gesamte Internet gelegt. Als Ergebnis von Projekt „Mocha“ wurde zusammen mit dem Netscape Navigator Version 2.02 eine Skript-Sprache für dynamische HTML Seiten der breiten Öffentlichkeit zur Verfügung gestellt. Diese Skript-Sprache hatte den Namen Netscape LiveWire. LiveWire wurde später umbenannt in JavaScript. Damit war ein Fundament geschaffen, auf welchem wir noch heute unsere Anwendungen aufbauen. Die Adaption von JavaScript verlief anfangs eher zögerlich. Hintergrund war der laufende Prozess der Sprach-Normierung und Weiterentwicklung gepaart mit den „Browser-Kriegen“ der damaligen Zeit. Es gab noch nicht viele Standards, bzw. wurden die Standards sehr hart erkämpft. Im Jahr 2005 erwähnte James Garret in seinem Aufsatz das erste Mal das Wort „AJAX“. Nun war es möglich, die dynamischen Inhalt im Browser auch mit Backend-Logik zu

koppeln, um noch interaktivere Anwendungen zu bauen. Die Beliebtheit von JavaScript stieg stetig an.

Neue Horizonte und Möglichkeiten

Wir spulen die Zeit nun etwas vor bis in das Jahr 2010. Das Internet ist überall. Auf dem Desktop. Auf unseren Smartphones. Auf embedded Devices. In der Gebäudeautomatisierung. JavaScript ist überall. Diese standardisierte Plattform ist eine ideale Grundlage für weitere Anwendungsfälle. In diesem Jahr betritt ein sehr interessantes Projekt die Bühne: Emscripten. Emscripten ist im Wesentlichen ein C bzw. C++ Compiler. Dieser Compiler erzeugt jedoch keinen Binärcode in Form von x86 Instruktionen, sondern er übersetzt das Programm nach JavaScript. Somit werden völlig neue Deployment-Szenarien für bestehende C++ Anwendungen möglich. Der Early-Adopter von Emscripten war die Entertainment Branche, voran das große Spiele-Studio Epic. Durch den Einsatz von Emscripten war es möglich, die Unreal Game Engine und bestehende Desktop Spiele mit minimalen Aufwand auch im Browser laufen zu lassen. Aus wirtschaftlicher Sicht war das ein gigantischer Sprung. Die Unreal Engine komplett in JavaScript neu zu schreiben hätte sicherlich ein großes Team über mehrere Monate bis Jahre ausgelastet. Durch Emscripten als Compiler wurde dies deutlich optimiert. Das Ergebnis war die Unreal Citadel Demo, welche im Browser lief. Als Laufzeitgeschwindigkeit wurde die halbe Framerate im Vergleich zu der Desktop Variante erreicht. Das Spiel war spielbar, jedoch gab es noch deutlich Luft nach oben.

Es geht noch schneller

Nach intensiver Zusammenarbeit zwischen Mozilla, Epic und dem Emscripten Team wurde im Jahr 2013 asm.js präsentiert. asm.js ist eine Untermenge von JavaScript, welche auf Laufzeiteffizienz ausgelegt ist. Durch die Annotation von Ausdrücken ist es möglich, der JavaScript Runtime Hinweise zu geben, um den Code effizienter auszuführen. Diese Typisierung von JavaScript macht es möglich, Integer und Fließkomma Datentypen genauer zu unterscheiden, und somit gerade die Integer Berechnung deutlich zu verbessern. Diese Unterscheidung ist in JavaScript nicht möglich, da es hier nur Fließkomma Zahlen gibt. Emscripten wurde mit asm.js als Compile-Target erweitert. Das Ergebnis war die Unreal-Engine mit der Cathedral Demo mit annähernd nativer Geschwindigkeit im Browser. Zu diesem Zeitpunkt gab es Browser mit sehr guten JavaScript Just-in-Time Compilern. Durch asm.js wurden noch zusätzliche Hinweise für die Optimierung des Laufzeitverhaltens

gegeben. Insgesamt war das Ergebnis im Sinne der Laufzeit sehr ansehnlich. Jedoch hatte das Ganze auch seine Schattenseite. Die Code-Basen wurden immer größer, der Quellcode immer komplexer. Zusammen mit der Größe und der Komplexität stieg natürlich auch die Download-Zeit für Anwendungen deutlich an. Und nicht nur die Downloadzeit stieg, sondern auch die Parsing-Zeit für JavaScript im Browser. Als Ergebnis ist die Time-To-Interactive für sehr große Anwendungen für spontane Internet-User nur noch bedingt akzeptabel. Die Suche nach einem Werkzeug, welches sowohl das Laufzeitverhalten als auch die Download- und Parsing Zeiten optimiert, begann.

Die Entstehung von WebAssembly

Im Jahr 2015 wurde als Reaktion auf diese Fragestellung die WebAssembly Working Group gegründet. Das Ergebnis dieser Working Group war ein Format, welches unterschiedlichste Anforderungen unter einen Hut bringt. WebAssembly ist Bytecode für das Web. Es handelt sich hier im ein binäres Format für transportable Programme. Dieses Format ist optimiert auf Download-Größe sowie Ausführungsgeschwindigkeit. Interessant an diesem Format ist, dass es in Host-Umgebungen integrierbar ist. Als primäre Host-Umgebung steht natürlich der Browser im Vordergrund. Es ist jedoch auch denkbar, WebAssembly im Backend zu nutzen. Es ist nur eine Host-Umgebung erforderlich, welche den WebAssembly Standard implementiert. Die Browser Hostumgebung setzt auf der bestehenden JavaScript Runtime im Browser auf. Durch diese Integration wird die bestehende Just-in-Time Compiler Mechanik im Browser wiederverwendet. Für die Browserhersteller wurde es so wesentlich einfacher gemacht, WebAssembly zu integrieren.

WebAssembly 1.0 MVP

Anfang 2017 war es dann soweit. WebAssembly wurde in Form eines Minimum-Viable-Product in Version 1.0 veröffentlicht. Bemerkenswert war, dass alle Browser Hersteller im Abstand von wenigen Tagen den WebAssembly Support ausrollten. Was ist nun Teil des MVPs? Schon im initialen Design wurden bewusst Stolpersteine ausgeklammert, um die erste Version des Produktes schlank zu halten. Jedoch wurde gleichzeitig auch dafür gesorgt, dass diese Abgrenzung keine größeren Nachteile verursacht. Beispielhaft beinhaltet die WebAssembly Host Umgebung eine Sandbox, in der das Programm abgeschottet läuft. Diese Sandbox ermöglicht den Zugriff auf einen linearen Speicherbereich. Dieser Speicherbereit

ist jedoch nicht managed. Der WebAssembly Autor muss das Speichermanagement und ggf. auch einen Garbage Collector selber implementieren. Dies ist jedoch kein größerer Nachteil, da WebAssembly ein Modul-System für dynamisches Linken beinhaltet. Es ist damit möglich, einen bestehenden Memory-Manager einfach in das WebAssembly zu linken und darauf aufzubauen. Ebenfalls wurde im MVP bewusst kein DOM Zugriff oder die opaque Datatypes implementiert. Es ist aus der WebAssembly Sandbox nicht direkt möglich, auf das DOM oder Browser APIs zuzugreifen. Auch dies ist auch kein größerer Nachteil, da über das Modul System dieser Zugriff über Wrapper-Types emuliert werden kann. Die Wrapper-Types verursachen zwar gewisse Laufzeit-Nachteile, diese sind jedoch im Vergleich mit der reduzierten Komplexität zu verkraften. Im MVP wurde bewusst auf Unterstützung für Threading und Thread Synchronisierung verzichtet. Dies ist einleuchtend, da die WebAssembly Host Umgebung auf der JavaScript Integration im Browser aufbaut. In JavaScript gibt es keine Threads und keine Thread Synchronisierung. Warum sollte also WebAssembly das in der Version 1.0 beinhalten? Dieser reduzierte Funktionsumfang ist jedoch kein Widerspruch. Durch die bewusste Ausklammerung dieser Funktionalitäten und gleichzeitiger Unterstützung eines Modul-Systems ist eine konsistente, nutzbare und vor allem erweiterbare WebAssembly Version 1.0 entstanden.

Kernkonzepte und Bootstrap

Ok, genug der Theorie. Wie funktioniert nun WebAssembly? WebAssembly selber kommt in zwei verschiedenen Ausprägungen vor, der textuellen und der binären Repräsentation. Die Textform(WAT) ist für uns Menschen gedacht, und soll uns vor allem beim Debuggen unterstützen. Das Textformat ist jedoch nicht besonders effizient für einen Computer lesbar. Deshalb gibt es das Binärformat(WASM), welches nur für den Computer gedacht und auf besonders effiziente Verarbeitung optimiert ist.

Diese beiden Formate sind sehr hardwarenah. Als Entwickler können wir solche Programme schreiben. Das wäre jedoch nicht wirtschaftlich. Der eigentliche Zweck von WebAssembly ist ein anderer. Als Entwickler schreiben wir Programme in einer Hochsprache, wie z.B. C++, Rust, .NET, Java oder auch Kotlin. Ein Compiler übersetzt diese Sprache dann nach WebAssembly. WebAssembly ist also primär ein Compile-Target für eine Hochsprache unserer Wahl. Wir haben somit das Beste aus zwei Welten: hohe Produktivität in einer bekannten Hochsprache und maximale Laufzeitoptimierung durch Einsatz eines optimierten Binärformats.

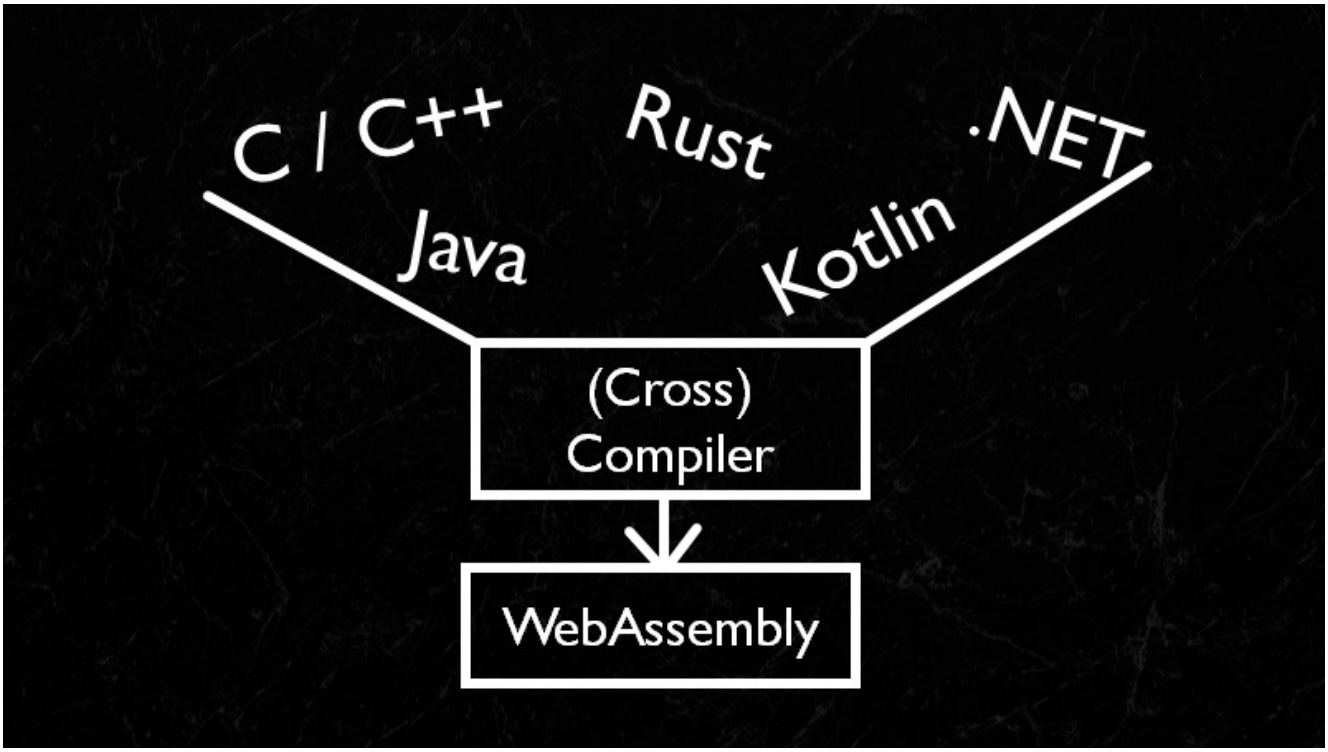

Abbildung 1: WebAssembly ist ein Compile Target

Wie funktioniert nun das WebAssembly Modulsystem? WebAssembly kennt sog. Module und Instanzen. Ein Modul ist eine Art Schablone, aus der mehrere Instanzen erzeugt werden. Jede Instanz bekommt ihre eigene Sandbox und kann darin ausgeführt werden. Das WebAssembly Modul hat eine Liste von Imports und Exports. Ein Export ist eine Funktion, welche von Außerhalb der WebAssembly Instanz aufgerufen werden kann, sprich aus der WebAssembly Host Umgebung. Der naheliegendste Anwendungsfall ist hier der Aufruf der Main Function des Programms. Interessant werden die Imports. Ein Import ist eine Funktion, welche von innerhalb der WebAssembly Instanz aufgerufen werden kann, jedoch nicht in der Instanz definiert ist. Importierte Funktionen werden z.B. für Interaktion mit der WebAssembly Host Umgebung benötigt. Jede Form von I/O wie eine Konsolen-Ausgabe oder eben auch ein DOM Zugriff wird üblicherweise in Form von Imports in das WebAssembly gelinked.

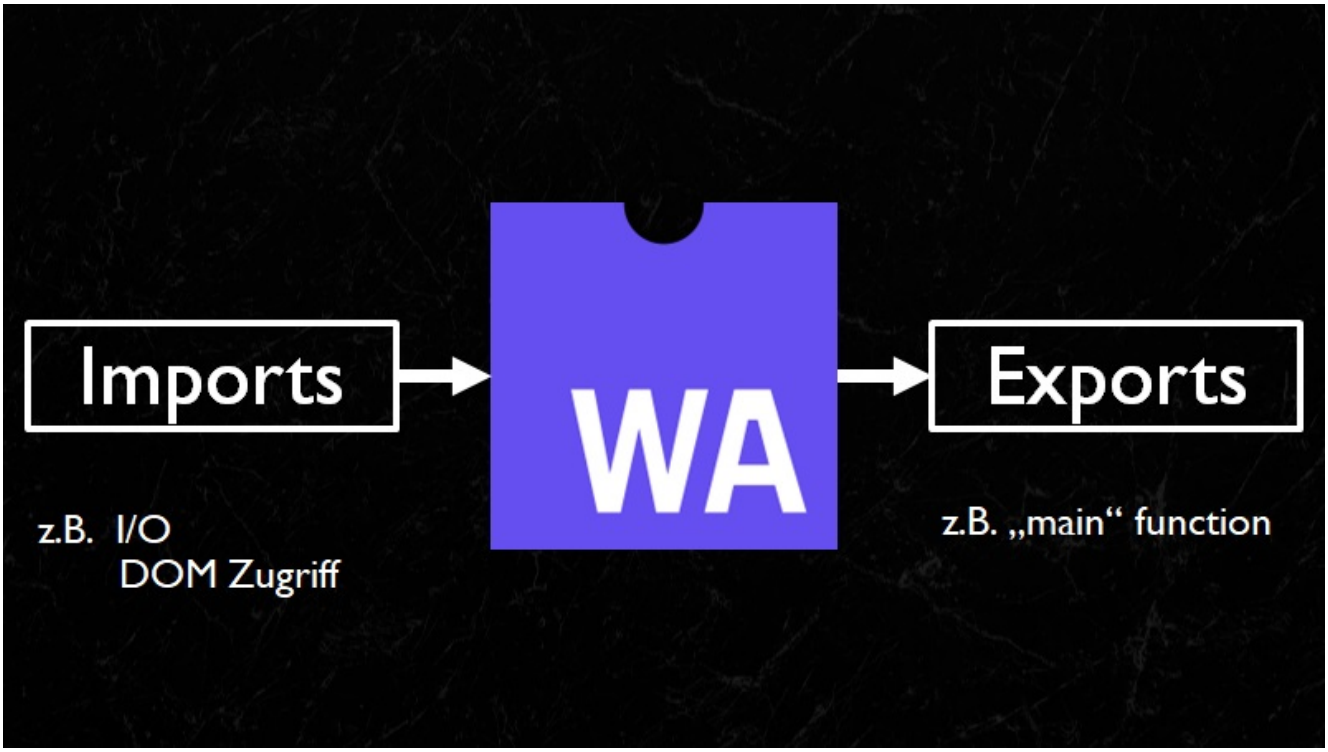

Abbildung 2: Das WebAssembly Modulsystem

Wie funktioniert nun dieses Linken? Um eine lauffähige WebAssembly Instanz zu erhalten, muss eine Bootstrap Sequenz auf dem WebAssembly Host durchlaufen werden. Im Browser gibt es hierfür eine JavaScript API. Im ersten Schritt muss die binäre WebAssembly Repräsentation geladen werden. Dies kann z.B. über einen XMLHttpRequest erfolgen. Wichtig hier ist, dass wir kein JSON oder XML Dokument als Antwort erwarten, sondern einen ArrayBuffer mit den geladenen Binärdaten.

```
var request = new XMLHttpRequest();
request.open('GET', 'bytecoder.wasm');
request.responseType = 'arraybuffer';
request.send();
```

Wenn nun das Promise fulfilled wird, können wir via WebAssembly JavaScript API die Instanziierung starten. An diesem Punkt müssen wir die Imports für das WebAssembly Modulsystem angeben. Ein fehlender Import führt zum Abbruch der Instanziierung.

```
request.onload = function() {
    var bytes = request.response;
    WebAssembly.instantiate(bytes, {
        // Imports
        mymodule: {
```

```
            add: function(a, b) {
                return a + b;
            }
        }
    });
};
```

WebAssembly.instantiate selber liefert ein Promise zurück. Bei der Instanziierung wird die WebAssembly-binäre Repräsentation validiert und kompiliert. Um den Main-Thread im Browser nicht zu blockieren, findet dies asynchron statt. Wenn dieser Vorgang abgeschlossen ist, wird das Promise mit den Referenzen auf das WebAssembly Modul und die WebAssembly Instanz fulfilled.

```
WebAssembly.instantiate(… ).then(function(result) {
    // Zugriff auf Modul und Instanz
    var wasmModule = result.module;
    var runningInstance = result.instance;
    // exportierte Funktion aufrufen
    runningInstance.exports.main();
});
```

Jetzt können wir den Kontrollfluss an die WebAssembly Instanz übergeben, indem wir z.B. eine exportierte Main Function aufrufen.

Lasst uns spielen!

Soweit, so gut. Welche Möglichkeiten habe ich, erste Erfahrungen mit WebAssembly zu machen? Zum Glück gibt es da eine sehr interessante Webseite: WebAssembly Studio. WebAssembly Studio ist JSFiddle für WebAssembly. Über diese IDE kann sehr einfach ein C oder Rust Projekt angelegt und dieses dann nach WebAssembly kompiliert werden. Der JavaScript Code für die Bootstrap Sequenz wird automatisch generiert und kann auch nachträglich editiert werden. Es ist auch möglich, die WebAssembly Text- oder Binärform genauer unter die Lupe zu nehmen und so tiefer in die Funktionsweise abzutauchen.

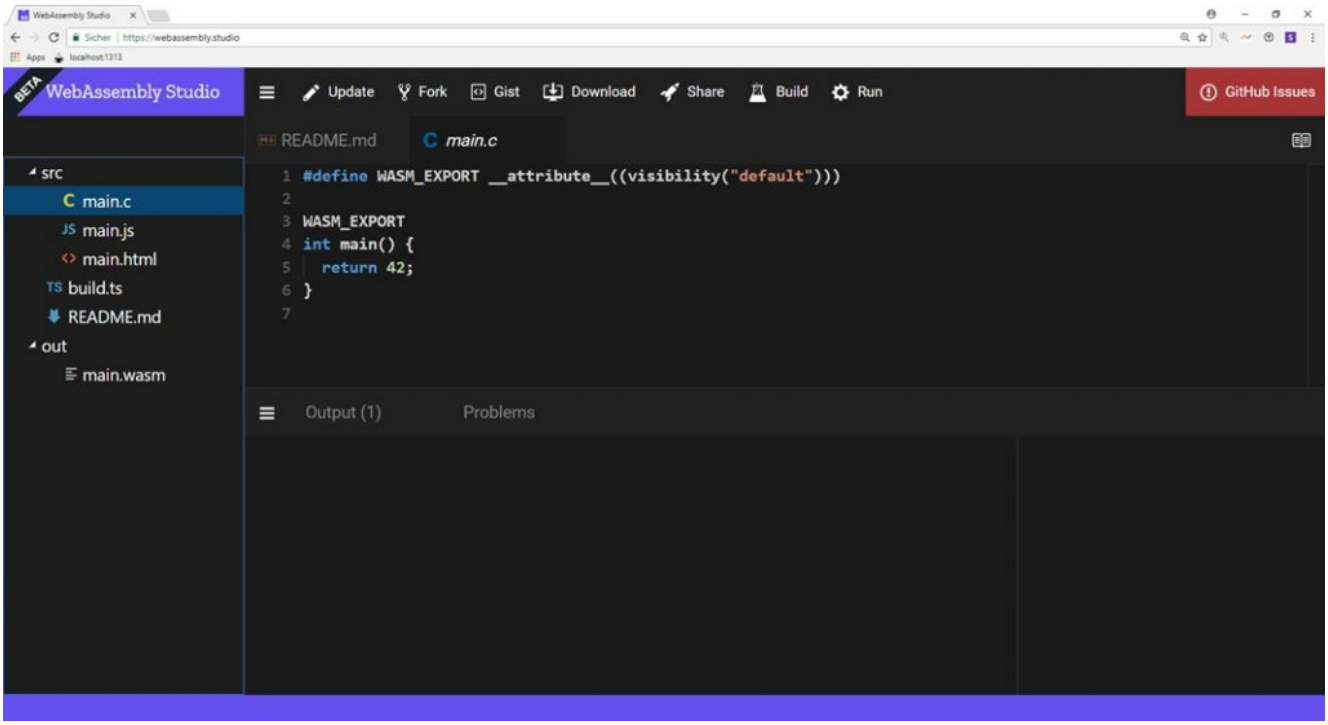

Abbildung 3: WebAssembly Studio

Wer nicht direkt Quellcode schreiben möchte, sondern eher eine lauffähige Anwendung benötigt, kann z.B. das Unity Tanks! Demo spielen. Hier handelt es sich um ein kleines Spiel, in dem Panzer in einer Landschaft bewegt werden können. Die Grafik ist sehr schön über WebGL umgesetzt. Hier zeigt sich sehr gut, wie WebAssembly bereits heute in Lösungen wie der Unity Game Engine integriert ist.

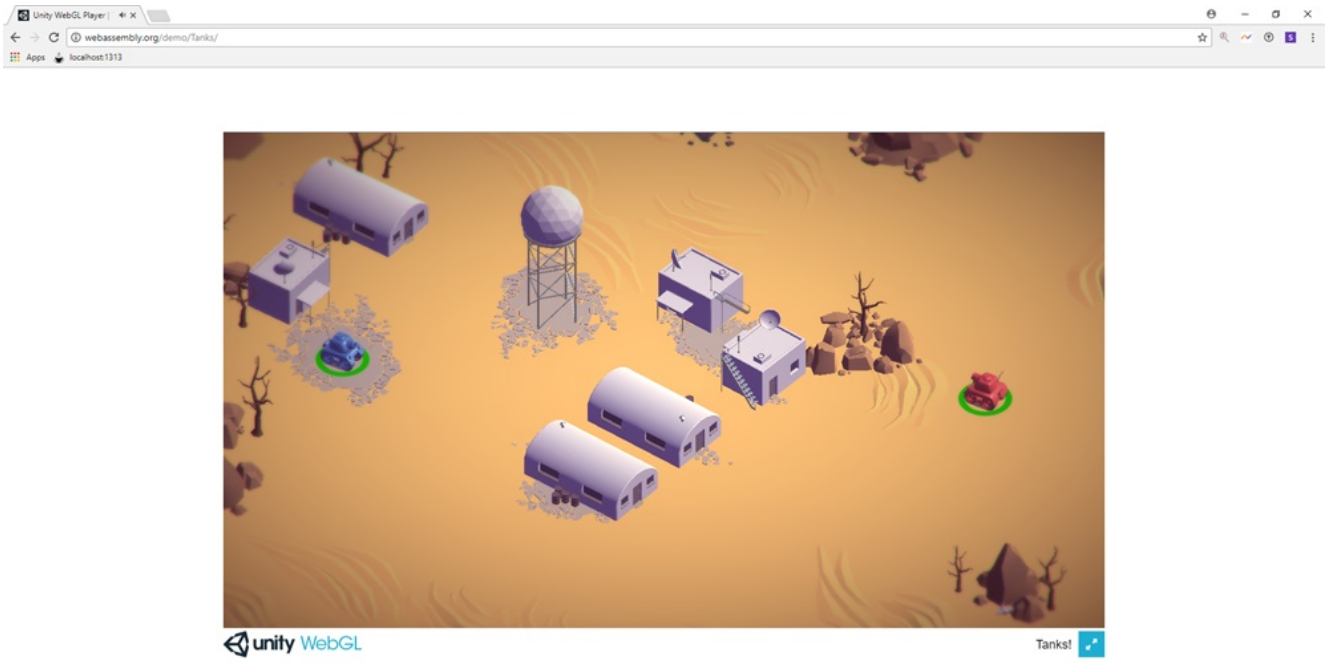

Abbildung 4: Unity Tanks Demo

Was mit der Unity Engine funktioniert, funktioniert auch mit der Unreal Engine. Das Sun Temple Demo zeigt sehr eindrücklich, wie WebAssembly und WebGL zu einem digitalen Kunstwerk verknüpft werden.

Abbildung 5 : Unreal Sun Temple Demo

Andere Anwendungsfälle

Es gibt natürlich auch Anwendungsfälle neben dem Gaming Bereich. Einer davon ist Pocketsphinxs.js. PocketSphinx ist ein System für Spracherkennung. Dieses System wurde via Emscripten nach WebAssembly übersetzt. Damit ist es möglich, Spracherkennung in Applikationen zu nutzen, ohne Remote-Services wie Alexa in eine Anwendung zu integrieren.

WebSight ist ein anderer Anwendungsfall. Hier wurde OpenCV via Emscripten nach WebAssembly übersetzt. Diese Demo greift auf die Webcam zu und sucht in dem Live Video Stream nach Gesichtern. Interessant hier ist, dass gleich drei Varianten der Bildverarbeitung in Form von WebWorker Instanzen parallel laufen. Eine WebAssembly Variante, eine asm.js Variante und eine JavaScript Variante. Hier offenbart sich der mögliche Performance-Gewinn deutlich: die WebAssembly

Variante ist um Faktor 15 schneller als die JavaScript Variante!

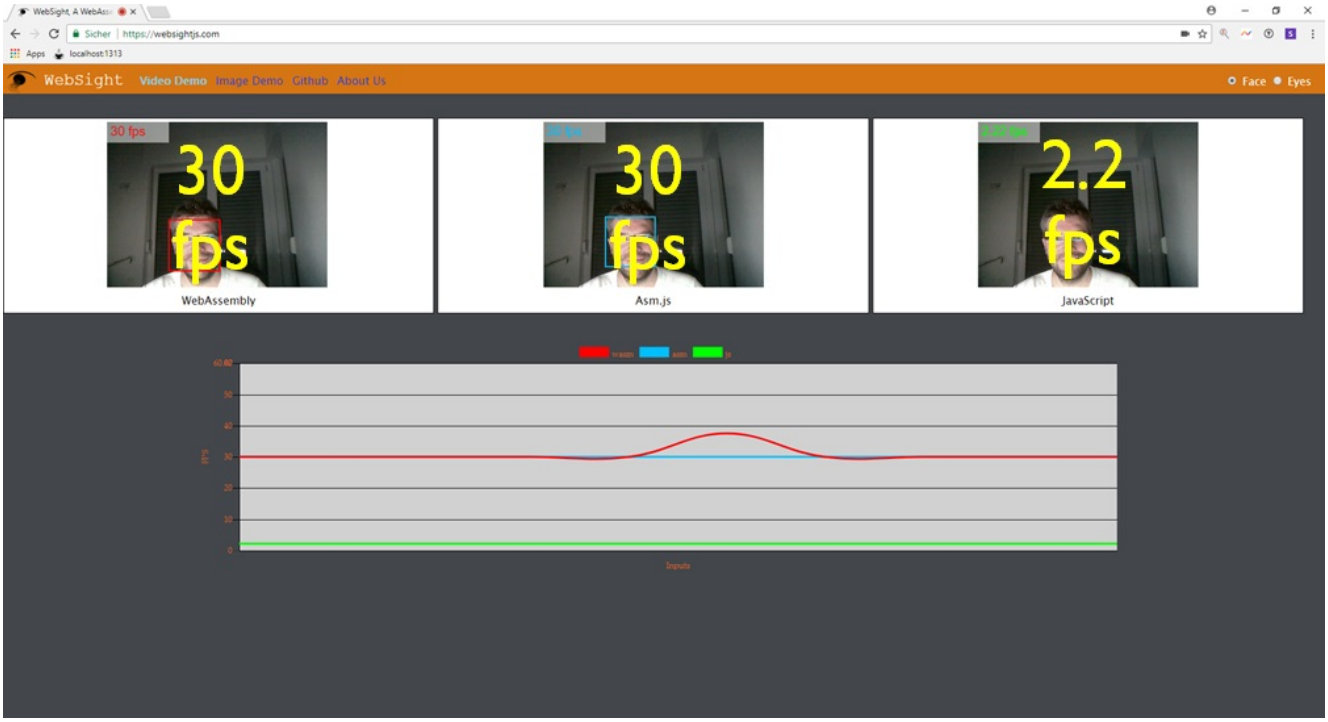

Abbildung 6: WebSight Demo

Missing Parts

Wo Licht ist, ist auch Schatten. Das WebAssembly MVP funktioniert, jedoch zeigen sich in der aktuellen Version besonders die Stellen, an denen bewusst abgespeckt wurde. Aus Sicht Entwickler fällt besonders das noch sehr einfache Tooling rund um die WebAssembly Integration im Browser auf. Es gibt im Moment nur sehr einfache Debugging Möglichkeiten. Die ersten Browser beinhalten Sourcemap-Unterstützung für WebAssembly, diese könnte jedoch noch deutlich optimiert werden. Durch das Fehlen eines Memorymanagers und einer Garbage Collection Integration muss diese Funktionalität via Modul-System importiert werden. In der Konsequenz verursacht dieser Runtime-Code ein unnötig großes WebAssembly Binary. Das Fehlen einer direkten DOM API oder des Zero-cost Exception Handlings im MVP verursacht ebenfalls Gluecode zur Emulation von Hochsprachen mit Exception Handling, welcher eigentlich nicht notwendig wäre. Unterstützung für Threading wäre wünschenswert, jedoch kann im MVP auch ohne native Thread Unterstützung arbeitet werden. Das WebSight Demo zeigt es: Threads können zusammen mit der WebWorker API emuliert werden. Dies ist für die meisten Anwendungsfälle eine gute Grundlage.

Ausblick

Auch wenn das WebAssembly MVP ein paar Ecken und Kanten hat, so ist es doch schon sehr gut einsetzbar. Die Unity Game Engine nutzt bereits heute WebAssembly für die Spieleentwicklung. Unity kann jedoch mehr. Denkbar sind hier alle Formen von besonders interaktiven Produktdemonstrationen, welche von WebAssembly profitieren können. Besonders in Verbindung mit Virtual Reality oder Augmented Reality ergeben sich völlig neue Anwendungsfälle. Eine weitere, interessante Möglichkeit ist die Migration von Legacy Code ins Web. Durch moderne Compiler können wir funktionierende und getestete Software in neuen Umgebungen laufen lassen wie zum Beispiel WebSight OpenCV auf einer Webseite. Die Möglichkeiten sind grenzenlos. Untermauert wird diese Perspektive durch die Übergabe der WebAssembly-Core-Spezification an das W3C. Diese Übergabe ist im Februar 2018 erfolgt. WebAssembly ist somit ein offizieller Standard mit unglaublich viel Potential für die Zukunft. Das Web war schon immer gut für Überraschungen und wird es mit dieser Technologie auch weiterhin bleiben!