Christoph Ernst Wottawa
MNr: 0527408
PR/Bachelor Thesis
30. September 2014

Medienprogrammierung auf Firefox OS

Was wird zur Entwicklung benötigt?

Benötigt wird zumindest ein Text Editor sowie der Firefox Web-Browser mit installiertem Firefox OS Simulator und ein Smartphone mit installiertem Firefox OS. Die Installation des Simulators gestaltet sich sehr einfach, da man die benötigten Komponenten als Add-Ons[1] herunterladen kann. Zum einfachen installieren und debuggen der entwickelten App ist außerdem das ADB Add-On hilfreich. Es ermöglicht bei angeschlossenem Gerät, im App-Manager des Firefox Browsers auf dem Desktop (im Menü unter *Extras* / *Web-Entwickler* / *App-Manager*) die Installation auf dem Gerät sowie das Debugging in Echtzeit.

In Firefox OS bestehen Apps aus HTML und Javascript. Damit das System eine App als solche wahrnimmt, benötigt diese eine Datei mit Namen „manifest.webapp“. Diese beschreibt wie die App heisst, von wem sie entwickelt wurde, welches Bild als Icon verwendet wird, sowie welche Privilegien sie hat und welche Permissions[2] die App benötigt (z.B. Zugriff auf das Adressbuch, oder die Kamera) Das Manifest wird in JSON geschrieben und sieht Beispielsweise so aus:

```
{
    "name": "Motion Detector",
    "version": "1.0",
    "type": "web",
    "description": "An Example for accessing the motion sensor in FFOS",
    "launch_path": "/index.html",
    "developer": {
        "name": "Christoph Ernst Wottawa",
        "url": "http://www.tuwien.ac.at"
    },
    "permissions": {
        "audio-capture": {
            "description" : "Needed for testing audio recording"
        },
        "video-capture": {
```

[1] https://developer.mozilla.org/de/docs/Tools/Firefox_OS_Simulator

[2] https://developer.mozilla.org/en-US/Apps/Build/App_permissions

```
            "description" : "with this enabled, we can try out video
recording"
        }
    },
    "icons": {
        "60": "/icon.png"
    },
    "default_locale": "en"
}
```

Diese manifest.webapp muss sich im selben Ordner wie restlichen Dateien der App befinden.

Bei einer Firefos OS App kann es sich entweder um eine „Hostet App“ handeln die, wie jede andere Website auch, von einem Webserver geladen und ausgeführt wird, oder aber um eine „Packaged App“, die als Zip File mit Manifest von einem Webserver oder aus dem Firefox Market geladen werden kann. Weiters werden Apps in drei Typen unterschieden. Es gibt „Web-Apps“, „Privileged-Apps“ und „Certified-Apps“. Nur packaged Apps können „privileged“ oder „certified“ sein. Außerdem ist der Typ „certified“ nur Apps von Mozilla und Geräteherstellern vorbehalten. Je nach Typ hat eine App verschiedene Zugriffsrechte auf JavaScript APIs des Systems.[3]

Anwendungsbeispiele

Widget-Based User Interface

Da das User Interface in gewöhnlichem HTML5 definiert wird, sind hier alle HTML Input Tags verwendbar. Im Beispiel wird ein <input type=„text“>, <input type=„date“> sowie zwei <button> Elemente verwendet. type=„date“ öffnet bei Firefox OS eine Datumseingabe. Auf die Eingabe kann mit JavaScript zugegriffen und diese beliebig verarbeitet werden.

```
//speichert die Input Elemente in Variablen
var txtInput = document.getElementById(„textInput");
var dateInput = document.getElementById(„dateInput“);
//liest den Wert der Input Elemente aus und speichert in Varbiablen.
var text = txtInput.value;
var date = new Date(dateInput.value);
```

Diese Zeilen liefern den Inhalt der Inputs zurück. Im Falle des Datums wird daraus direkt aus dem input.value ein Date Objekt erzeugt.

[3] https://developer.mozilla.org/en-US/Apps/Build/Manifest#type

Canvas-Based User Interface

Zum zeichnen von graphischen Elementen gibt es in HTML5 das <canvas> Tag. Um in ein solches Canvas zu zeichnen benötigt man einen Kontext. Alle Zeichenbefehle werden auf diesem Kontext ausgeführt.

Die draw() Methode löscht zuerst den Inhalt des Canvas und zeichnet danach ein Rechteck mit einem Eckpunkt an der Stelle des Touch-Events.

HTTP Connection

HTTP Verbindungen mit JavaScript herzustellen ist schon seit einiger Zeit mit Hilfe von XMLHttpRequest möglich und wird in dynamischen Websites benutzt um Daten im Hintergrund nachzuladen, ohne die ganze Seite neu zu laden. Mit XMLHttpRequest ist es möglich, beliebige textuelle Daten, wie XML oder JSON aber auch binäre Daten wie Bilder, zu übertragen. Dabei ist sowohl das Empfangen als auch das Senden möglich - es können also auch Uploads durchgeführt werden.

Zu beachten ist dabei, dass HttpRequests normalerweise nur an den Server, von dem die App stammt, gerichtet werden dürfen. Möchte man Daten von einem beliebigen anderen Server laden, muss der App Typ auf „privileged“ gesetzt werden und die Permission „SystemXHR“ im Manifest vermerkt sein. Andernfalls wird der Zugriff blockiert.

Personal Information Management

Der Zugriff auf das Adressbuch ist recht einfach zu bewerkstelligen. Die App muss dazu vom Typ „privileged“ sein und benötigt die Permission „contacts“.

```
"permissions": {
      "contacts" : {
            "description" : "Needed for Testing!“,
            "access": "readcreate"
      }
}
```

Mit „new mozContacts()“ kann man einen neuen Kontakt erzeugen, welcher mit Hilfe des Contact Managers „navigator.mozContacts“ im Adressbuch speichern kann. Zu beachten ist, dass das mozContact Object alle Felder als Array speichert.

Das API ist auf den Seiten von Mozilla unter Contacts API[4] beschrieben.

Geolocation

Das Anfordern einer Geolokation erfordert keine Privilegien und kann daher von jedem Typ von App erfolgen. Der Benutzer wird vom System gefragt, ob er seinen Standort freigeben

[4] //https://developer.mozilla.org/en-US/docs/Web/API/Contacts_API

will. Die Abfrage erfolgt mit Hilfe einer watchID, wenn man die Position auf längere Zeit verfolgen möchte:

```
var watchID = navigator.geolocation.watchPosition(printPosition, error,
options);
```

Sobald eine Position verfügbar ist, wird printPosition aufgerufen und die Position übergeben. Falls ein Fehler auftritt wird der Errorhandler ausgeführt. „options“ legt Optionen für die Positionsbestimmung fest.

SMS/MMS

SMS und MMS können derzeit nur von „certified“ Apps verschickt werden. Dies betrifft allerdings nur veröffentlichte Apps. Auf dem Testgerät funktionierte es trotzdem.

Wenn man SMS/MMS auch ohne zertifizierter App verschicken möchte, gibt es noch die Möglichkeit WebActivities zu benutzen. Diese binden dann die Funktion des Systems in die eigene App ein. Der Beispielcode enthält die entsprechende Fallback Methode.

Das API ist auf den Seiten des Mozilla Developer Networks dokumentiert.[5]

Bluetooth

Zu Bluetooth konnte kein funktionsfähiges Beispiel erstellt werden. Laut Dokumentation[6] ist auch dieses Interface nur zertifizierten Applikationen vorbehalten. Jedoch konnte weder im Simulator, noch auf einem Testgerät mit Firefox OS 1.3 ein Zugriff auf den Bluetooth Controller über „navigator.mozBluetooth“ erlangt werden. Die Dokumentation dazu ist außerdem noch als Draft markiert.

WebGL

Mit Hilfe des WebGL Standards in HTML5 ist es mittlerweile in den meisten Browsern möglich, hardwarebeschleunigte 3D Grafik zu erzeugen. So funktioniert dies auch unter Firefox OS. Ein gutes einführendes Tutorial zum Theam findet sich auf den Seiten des MDN[7]. Auf dieser Anleitung beruht auch der Code des Beispiels.

In HTML wird ein Canvas definiert, in dem der WebGL Inhalt gezeichnet werden soll. Dann wird versucht einen WebGL Kontext zu bekommen in dem dann mit WebGL gezeichnet wird.

Um zeichnen zu können, müssen in WebGL immer ein Fragment- und ein Vertex-Shader geladen werden. Diese werden in GLSL geschrieben und befinden sich in unserem Fall

---

[5] https://developer.mozilla.org/en-US/docs/Web/API/WebSMS_API

[6] https://developer.mozilla.org/en-US/docs/Web/API/Web_Bluetooth_API

[7] https://developer.mozilla.org/de/docs/Web/WebGL/Einf%C3%BChrung_in_WebGL

innerhalb des HTML Quellcodes, als <script> Elemente. Sie werden mittels des ID Attributes aus dem HTML Code ausgelesen. Shader müssen zu einem Programm kompiliert und verlinkt werden, was in der initShaders() Methode passiert.

Das zu zeichnende Objekt wird in der Methode initBuffer() in einem Array, dem Vertex Buffer, als dreidimensionale Koordinaten gespeichert.

Im Gegensatz zu OpenGL gibt es in WebGL keine eigenen Methoden zur Rotation oder Transformation, wie z.B. glRotate(). Diese Funktionalität muss entweder selbst geschrieben werden, oder entsprechende Hilfsbibliotheken eingebunden werden, die den Umgang mit den erforderlichen Matrizen wesentlich vereinfachen. Im Codebeispiel wird Sylvester.js[8] sowie die aus unbekannter Quelle stammende glUtils.js eingesetzt.

Schlussendlich sind in der drawScene() Methode alle restlichen, zum Zeichnen benötigten Routinen zusammengefasst. Die Szene wird komplett gelöscht und danach anhand einer einfachen Perspektivmatrix gerendert. Die Model View Matrix wird mit einer Identitätsmatrix initialisiert, welche bei jedem weiteren Zeichenvorgang entsprechend der gewünschten Rotation neu berechnet wird. Dabei wird die Rotation immer anhand der ursprünglichen Model View Matrix berechnet, vor dem Zeichnen auf einem Stack gespeichert um danach wiederhergestellt zu werden.

Die Zeit zwischen dem Neu-Zeichnen wird mittels timestamp aufgezeichnet um so die vergangene Zeit festzustellen und die Rotation entsprechend berechnen zu können. Es kann also eine beliebige Framerate durch Aufruf von setInterval() erzeugt werden, die Rotation läuft immer gleich schnell ab.

Media Playback

Um Video oder Audio abzuspielen, genügen lediglich das entsprechende Tag im HTML Markup (<video> oder <audio>). Wenn das Attribut „controls“ angegeben wird, werden Steuerungselemente zum Video eingeblendet. Im Beispiel wird das Video per JavaScript Code gestartet.

Audio Recording

Der Zugriff auf das getUserMedia()[9] Interface ist zwar allen drei App Typen möglich, jedoch wird die Permission „audio-capture“ im Manifest benötigt. Der Benutzer muss den Zugriff auf das Mikrofon bestätigen.

getUserMedia stellt einen Stream zur Verfügung der in diesem Fall für Audio konfiguriert wurde. das MediaRecorder Objekt ermöglicht die Aufzeichnung des Streams und gibt beim Beenden der Aufnahme einen Blob, ein Binary Large Object zurück, welches an das

[8] http://sylvester.jcoglan.com/

[9] https://developer.mozilla.org/en-US/docs/NavigatorUserMedia.getUserMedia

<audio> Element im HTML Dokument zur Wiedergabe übergeben wird. Außerdem wird noch ein Link zum „Download“ der Datei erzeugt, der es ermöglicht, die Datei zu speichern.

Video Recording

Das Aufzeichnen von Video der Webcam funktioniert im Prinzip genau wie im vorigen Beispiel mit den Audio Daten. Einzig das Manifest der App muss auch die Permission „video-capture“ beinhalten. Die Funktion ist erst ab Firefox OS 1.4 implementiert.

Snapshot Image

Zum Erstellen eines Snapshots aus dem Video-Stream der Kamera des Geräts, greifen wir wieder auf getUserMedia() zu. Der Stream wird diesmal direkt an ein <video> Element weitergeleitet, sodaß ein Live-Video angezeigt wird. Um ein einzelnes Bild aus dem Video zu speichern, wird dieses in ein verstecktes <canvas> Element gezeichnet, welches die in ihm gezeichneten Inhalte als Bild ausgeben kann. Diese Bilddaten werden mittels eines <img> Elements angezeigt.

```
canvas.toBlob(function (blob) {
      file = blob;
}, "image/jpeg", 0.95);
```

Durch die Methode toBlob() wird aus dem Inhalt des Canvas eine Bilddatei, die über das getDeviceStorage() Interface in der Bildgalerie des Gerätes gespeichert wird. Dazu ist muss die App „privileged“ sein und im Manifest die Zeile

```
"device-storage:pictures" : { "access" : "readcreate" }
```

enthalten sein. Sie ermöglicht es Bilder zu lesen und zu erstellen.

Image Processing

Wie im vorangehenden Beispiel werden Video-Daten durch getUserMedia() eingelesen. Jedoch wird das dazugehörende <video> Element nicht angezeigt, sondern dient lediglich dazu, die Bild Daten aus dem Video auszulesen und in ein Backing-Canvas und -Context zu schreiben. Der Context ermöglicht den die Byteweisen Zugriff auf die Bilddaten, sodaß ein Threshold Filter angewandt werden kann. Erst nach der Filterung werden die Daten an das sichtbare Canvas geschickt. Dies geschieht alle 40 ms, sodaß sich eine Framerate von 25 Bildern pro Sekunde ergibt.

Motion

Die Spezifikation dieses Interface ist auf den Seiten von Mozilla derzeit noch als „Working Draft“ gekennzeichnet. Gleichermaßen wird dieses API auf einem Testgerät mit FFOS 1.3 nicht unterstützt, obwohl dieses einen entsprechenden Sensor enthält. Die Funktion des

Codes konnte jedoch im Google Chrome Browser auf einem MacBook Pro (diese enthalten einen Bewegungssensor, den Chrome unterstützt) überprüft werden.

Der Zugriff erfolgt indem ein Eventhandler an das Ereignis „devicemotion“ gebunden wird. Jedes Mal wenn also neue Bewegungsdaten verfügbar sind, wird dieser aufgerufen und aktualisiert die Daten im User Interface.