WebXR or WebVR 2.0

WebXR is an evolutionary development of the old WebVR APIs that addresses user feedback and additional use cases that were not available when the orginal WebVR apis were created.

The Immersive Web Community Group expects the final WebVR compatible draft to be released this summer (2019) and the candidate recommendation with two interoperable implementations to be released towards the end of the year.

While we wait for the WebXR specification to solidify, we can play with libraries that abstract some of the underlying complexities of WebVR, WebXR and the basic Three.js library.

Google `<model-viewer>`

An interesting tool I saw at SFHTML5 is `model-viewer`, a web component-based tool that will display 3D content in a 2D hosting page.

`model-viewer` provides a way to view 3D content in a web browser, independently of the form factor you're using (either a 2d Desktop browser or the browser built into a 3D device like the Oculus Quest). It is a web component so we need to polyfill web components for older versions of Edge and Firefox.

In the video below, Chris Joel, from the DeviceXR at Google explains `model-viewer`, what it is and how it works.

This assumes that we've loaded the polyfills using NPM. This is not an absolute requirement. You can always load the script from a CDN but, I prefer to have the scripts in my local directory

```
npm i @google/model-viewer \
@webcomponents/webcomponentsjs \
fullscreen-polyfill \
intersection-observer \
resize-observer-polyfill
```

I normally put the scripts at the bottom of the page, before the closing body tag and before loading the model viewer scripts.

```
<!--
webcomponents-loader is required to support older
versions of Edge and FIrefox
-->
<script src="./js/webcomponents-loader.js"></script>
<script src="./js/intersection-obs<script src="./js/webcomponents-loader.j
</script><!--
Fullscreen polyfill is required for experimental
AR features in Canary
-->src="./js/fullscreen.polyfill.js"></script>-->
```

```
<!--
Magi</script><!--
Magic Leap support
-->/prismatic.js"></script>`</pre>
</script></script>`</pre>
```

After we load the polyfills, we can load the `model-viewer` scripts. We use modules to load the scripts in modern browsers and `nomodule` to load a different script for those browsers that don't support modules.

```
<script type="module" src="./js/model-viewer.js"></script>
<script nomodule src="./js/model-viewer</script>s"></script>
</script>
```

Once we have all the scripts installed we can use `model-viewer` by creating one or more instces of the custom element like show in the example below:

```
<model-viewer
  src="./models/2CylinderEngine.gltf"
  alt="A 3D model of a 2 cylinder"
  auto-rotate
  camera-controls></model-viewer>
```

You can have as many instances of `model-viewer` as you need on your page. They can each be customized independently from each other. The two additional instances have different background colors. They can also be manipulated separately.

We can also load multiple models

But the more intriguing, to me, thing about `model-viewer` is how, with a few extra attributes you can provide a range of experiences, from mouse manipulation as show in the prior examples to full support for Google's ARCore, Apple's Quicklook (with some additional requirements), and Magic Leap.

For example, adding the `ar` attribute will provide a way to launch Google's

Scene Viewer in supported devices. See Using `model-viewer` to launch Scene Viewer for more information.

The code now looks like this:

```
<model-viewer
  src="./models/Satellite(1).gltf"
  alt="3D Satellite"
  ar
  camera-controls
```

The model won't look any different in desktop or devices where Scene Viewer is not supported but, in devices where the feature is supported, users will have an option to click on to place an interact with the object.

Apple provides its own AR experiences on supported iOS 12+ devices via AR Quick Look on Safari. This requires an additional USDZ model which will be used only in supported iOS devices.

If you have the USDZ model, you can add it to your experience using the `ios-src`attribute and the URL to the model.

```
<model-viewer
  src="./models/Diner.gltf"
  alt="3D Diner"
  camera-controls
  ios-src="./models/Diner.usdz"></model-viewer>
```

And, of course, you can provide support for both, using `ar` and `ios-src` together to cover both Android and iOS in the same element

```
<model-viewer
  src="./models/Diner.gltf"
  alt="3D Diner"
  camera-controls
  ar
  ios-src="./models/Diner.usdz"></model-viewer>
```

Final Notes

This barely begins to scratch the surface of what we can do with `model-viewer`.For more information see the [model-viewer documentation and this 2018 post about Augmented reality for the web that talks about the ideas that led to

Usdz is the wrinkle in the process. Most of the models in this page are loaded from glTF files, an interchange format created by the Khronos Group and that has wide support in terms of tools and technologies. As pointed out in All about Apple's new USDZ File Format—Simplified the technology was developed by a single vendor, and has no open source tool support but there is commercial support, bot from small companies like Vectary and large tool companies like Adobe with Project Aero.

For more information about UDSZ, you can check the following resources.

- Usdz File Format Specification
- All about Apple's new USDZ File Format—Simplified
- Apple's USDZ AR file format: What you need to know
- What is USDZ and why you should care - Apple's AR kit explained

FInally, I've made a copy of this page in Github where you can play with the working demos and test with Android and iOS: https://caraya.github.io/model-viewer-demo/index.html