

WebAR for CIFT

Felix Kopp, Mat. no. 6290790, 2020-01-17, Augmented Reality PStA

Abstract

This application, accessible under <https://webar.sandtler.club/>, is an augmented reality application designed for smartphones. It renders an arbitrary set of vectors received over a REST API in almost real time over a Hiro AR marker. The generated documentation is browsable under <https://webar.sandtler.club/apidoc/>.

General Architecture

The application is written in TypeScript¹, a subset of JavaScript that extends the language with features like interfaces and strong typing which makes it easier to maintain in the long term. In order to keep page load times at a minimum, all code is bundled and minified using Webpack². AR marker recognition is realized with the ar.js³; a library that, besides being poorly written, has excellent performance as well as native three.js⁴ integration. The latter is used for rendering vector data that are streamed from the server in the browser viewport.

The rendering loop uses a basic layer of abstraction, which is provided by the `AbstractRenderModule` class. Among other features, it has callbacks for initialization, painting, and canvas resizing that can be hooked into with ease. The `ARRenderer` class is supervising all modules and initializes the context for three.js. It serves as the main outward-facing API, if this code base were used as one. New `RenderModules` can be added dynamically to it via the `addModule()` function. Every new module is instantiated by the control code that also spins up the `ARRenderer` class. In this project, it is located in the `main.ts` file. This makes it easy to pass configuration parameters to the module's constructor and thus allows for a high degree of customizability.

¹ <https://www.typescriptlang.org/>

² <https://webpack.js.org/>

³ <https://github.com/jeromeetienne/AR.js>

⁴ <https://threejs.org/>

Up to this point, everything described is working in sync with the renderer, making development remarkably straightforward. However, retrieving the vector data in almost real time meant asynchronous network operations were necessary. To cope this, a simple buffering mechanism was implemented with the `VectorDataSource` class. New data is pulled in bulks of configurable size, and when the buffer length falls below a certain threshold, it is refilled. While the networking requests itself are async, the `VectorFieldRenderModule` can still pull new data in sync with the main rendering loop. If the buffer ever underruns, the corresponding method call will just return `null` which in turn tells the render module to not update the screen content.

The AR.js framework hooks directly into three.js, which is why `ARRenderer` exposes some of its elements with getters. These are then accessed by the `ARRenderModule` and passed to the framework upon initialization, acting as a convenient wrapper for AR.js.

Challenges and Pitfalls

Perhaps the most notable difficulty was time. The project was started 12 hours before submission, requiring many design choices to be left to intuition. Fortunately, this strategy turned out successful in most cases. In some instances, however, it demanded hours of time. The most notable example is using Webpack and therefore NodeJS modules: Even though most libraries behaved well and worked without a problem, the AR.js library seemed to rely heavily on being loaded as an external script in a browser rather than bundled into a single file at transpile time. The solution was the `script-loader`⁵ package from npm, providing a compatibility wrapper for this exact scenario. Furthermore, the AR.js library was poorly documented, even requiring manual code inspection to understand how it should be used in some cases. The modularized design helped a lot, however, allowing the wrapper code to be written once and then never touched again.

⁵ <https://www.npmjs.com/package/script-loader>