Wirtual

Next Generation User Interfaces

Final Report

Oguz Gelal
Subhani Shaik
Raphael Duhen

1 - Introduction

1.1 - Abstract

This report is about Wirtual, a front-end library written in javascript that brings virtual reality to the web. In this section, we will be first giving a general overview on what Wirtual exactly is. We will tell what it does, and how it does it without going into deep technical details. More technical details will be on chapter 3 - Technical Details. We will briefly specify the intended problems that we addressed, and whether we solved it or not. We will inform you shortly about the objectives we had in order to solve these problems. Last but not least, we will specify the methodology we have used to fulfill these objectives, and much more will be on the Technical Details chapter. We will also be talking about what has been done before within the same direction with Wirtual, and how our library is different from them and what are the values that we adding over those solutions.

Wirtual is a library for developers, which has the goal of helping them build 3D-VR products for end users. Therefore, we have two target audiences. Our direct audience are developers, and indirect (second level) target audience are the end users. More on the target audiences could be found in section 2.1, target audience analysis. Throughout this report, whenever a target audience analysis is needed, the analysis for both target audiences will be mentioned.

1.2 - Overview

Wirtual is a javascript library that has the goal of bringing the world of virtual reality to the world wide web. The main philosophy of this library is its ease of use and completeness; that is, to make developers life extremely easy and simple without sacrificing from features, scalability and usability.

With Wirtual, developers can write pure HTML & CSS, easily integrate the things they built into the 3D-VR world, manipulate it within this world, and do all this using nothing but HTML. Moreover, it comes with a variety of very well-known and widely used tools that the developers love and work with daily, and that users experience on every corner of the web, with or without realising it; tools such as jQuery [1], Twitter Bootstrap [2], and animate.css [3]. Therefor, Wirtual creates an environment for developers to create products without them having to relearn web development, and gives users products they love and are already familiar with - only in virtual reality and enhanced with its capabilities. Apart from that, Wirtual allows developers to use the modern technologies of the web along with it. Libraries such as AngularJS, React, Meteor etc. will work in harmony with Wirtual, allowing developers to build any kind of application they want.

Providing all these advantages, it does not sacrifice from scalability. In fact, Wirtual supports a wide range of cases, from high-end VR glasses with expensive equipment to the cheapest of

Google Cardboard devices, from mobile phones to desktops with old browsers. For mobile, it makes use of the gyroscopic sensors and touch spanning. The library relies on WebVR api, which is a standard, so users browsing on computers can see it on their screens. WebVR api is an experimental feature and is not supported by any modern browsers latest stable version. That's why, Wirtual utilizes a polyfill for being able to support older versions of browsers. All these support comes built-in, without developers having to do anything special.

Wirtual has a variety of drop-in components / functionalities which developers can use only by adding a class / tag to their HTML codes. We call these features the HTML api, and that is how it will be addressed throughout this documentation. HTML api makes the library much more developer friendly and understandable - for instance the 3D grid system makes it a breeze for developers to understand and place an item anywhere in the 3 dimensional space. Wirtual's usage however is not restricted to what it can offer with its HTML api. Wirtual has a JS api attached to the *window* object, under the namespace *Wirtual*. This means the JS api is global and is accessible anywhere within the code, and could be used regardless of the context. The JS api gives developers a set of useful functionality for manipulating their objects programmatically. If developers want further and more advanced usage, the JS api exposes the instances of Three objects for developers to interact with them independently from Wirtual. Wirtual is powered by a very sophisticated and powerful library called Three.js [4]. Three.js is widely used in its area, and has a huge community. A huge community means a well-prepared documentation, lots of tutorials and a lot of useful resources on the internet. So Wirtual developers having the ability to access Three objects through our api allows them to take advantage of all these documentation and useful information that are available on the internet.

Along with all the advantages to developers, it comes with features to make it more usable and pleasurable for the users too. One example to this is the virtual keyboard. Since users wearing a VR headset cannot see the actual keyboard, a virtual one pops up from the bottom of the page, which allows them to type with mouse clicks. Another example would be the position tracking on mobile phones - users are able to look around by turning their phone (instead of only swiping & touch controls).

1.3 - Problems addressed

1.3.1 - Developer related

- **Difficulty** - 3D and/or VR development is hard. It is complicated. It requires advanced knowledge of applied mathematics. It requires the knowledge to use specific engines, such as Unity or Unreal Engine; or worst, it requires knowledge of quite low level API's such as OpenGL or WebGL. For most cases, it even requires the knowledge of 3D modelling.

- **Learning curve** - As the mentioned cases above, it requires quite specific knowledge to build 3D-VR content. A developer cannot jump in coding without spending at least months of self-training.
- **Time Consumption** - Along with the expertise it take, building a 3D-VR project from scratch will take a whole lot of the developers time due to the reasons above.
- **Not for web** - VR products are not meant to work for web. Usually they are graphics oriented, and are in the form of games or 360° videos and / or panoramic images. They don't support common user interactivity elements such as forms and text inputs. It is almost impossible to build a decent web page which gives user text-based information and basic level of interactivity.
- **Scalability** - Scaling a 3D-VR application is extremely hard. Even though some engines like Unity supports an array of VR glasses, they fall short on web and mobile web environments - their support for WebGL and VR are mutually exclusive and they cannot export to web with built-in functionality [5]. Moreover, some solutions for web such as aframe.io will fall short on non-VR-ready browsers [6].
- **HTML Rendering (CSS3D Support)** - Rendering HTML along with 3D content, especially in VR, is not only not supported by any 3D-VR web solution, it is also not accurate and practical, and very hard to utilize.
- **Maintainability** - A well-organised codebase will be easy to maintain. However, with all the advanced techniques and methodologies, it is hard to keep the code organised, and therefore to maintain.

1.3.2 - End-user related

- **Need for glasses** - For experiencing any VR content, users need to have at least a Google Cardboard device, or they will have no idea on what the content looks like. There is no fallback for at least let them get the feeling through their screen.
- **Need for experimental browsers** - For the 3D-VR contents on the web that could actually work on browsers, there are no support for non-VR-ready browsers. Many users do not have experimental browsers installed, and will not do so to just to view a content.
- **Limited contents** - Web aside, there are not too many 3D-VR content on the market. These do not touch many users, as the percentage of VR users is as small as 16% worldwide [7]. Most of these content that exists are gaming related, and are based on graphics / multimedia. Ones that actually work for the web are also mostly multimedia related, almost no 3D-VR application on web takes advantage of web elements to build an interactive and innovative content. This is a result of 3D-VR web being a new and experimental field, therefore lacking compatibility and adequate development tools.

1.4 - Objectives

With Wirtual, we aimed to build a next generation user interface for developers (as users), and for end-users to address the problems mentioned in section 1.3.

For the developers;

- We wanted Wirtual to be an easy drop-in solution for 3D-VR development, that developers can build their VR applications without committing long hours for development and training for mastering difficult tools and technologies. In fact, we wanted them to be able to build VR content with what they already know - with simple HTML & CSS.
- As opposed to other 3D-VR web solutions, we wanted Wirtual to be much more scalable, easy to use, and high level (providing drop-in components instead of allowing developers to draw cubes, spheres, lines etc.).
- All other 3D-VR solutions focuses on creating graphic-oriented, usually gaming or multimedia content, whereas we wanted Wirtual to focus on creating more informative / interactive products.
- We wanted the library to allow 3D model files to be imported with a few lines of HTML code.
- We wanted the library to handle skyboxes, lights, and all other complications of building a 3D scene, eliminating them from developers concern.
- We wanted a library that has built-in support for HTML rendering - for allowing developers to build their components easily.

For the end-users;

- Rather than solving a problem, we first wanted to bring a pleasant upgrade to the web to enhance users experience, taking advantage of virtual reality and all its capabilities.
- We wanted to make VR accessible for as many users as possible.

And for all the internet community, we wanted to improve the variety of applications available. Since this is a rather new field, there is not too many examples of applications available. Especially with easy HTML Rendering and other wide range of support and compatibility we aimed to provide, the possibilities are endless. We want developers around the world to be able to kick-start their ideas in VR. We wanted to make our share of contribution to this newly rising technology that is still in research and development. Last but not least, since it is very simple to build VR experiences with Wirtual, we intend it to be used for educational purposes; more information on this will be given in section 2.1.1 Target audience analysis - developers.

1.5 - Methodology

This section is where we explain the how we executed the necessary actions in order to address all the problems we mentioned, and fulfill all the promises we made. The steps we followed will be briefly explained, but not in much technical detail.

The first step was the learning process. We were no experts on 3D-VR development even though we had previous web experiences. After a lot of reading and researching, we decided that Three.js is the perfect library to power Wirtual. Then we spend some time training ourselves in Three.js. Once we are comfortable with it, we started of by crafting everything manually. We manually included all the necessary libraries in an html file, and made a basic scene with only a sphere in it. We spent a lot of time manually putting together all the libraries to create an actual VR environment. When we managed, we created the actual codebase. We created the build system, which downloads the right versions of these libraries, merges all into a single file and produces an output. We tested the output file with the sphere we already had. Once confirmed working, we started to build the compiler. Compiler is the most sophisticated part of this library. We first get the compiler to render the HTML and detect the *wr-** classes and tags. We then started building the components and making the compiler utilize them when necessary tags / classes are used. Our first component was the scene. Second was the lights. Third was the spinning earth. That's when we had an actual 3D-VR scene with an object in it, so we focused on the change detection algorithm. We spent a lot of time on this. After this is done, we started doing the 3D grid system. This required some trigonometry, gave us a hard time. In the meantime we also executed our plans on the data flow - about how the components will exchange data with each other. After 3D grid system and a bunch of other Three.js components are done, we started to build the HTML renderer (the one with CSS3D support that renders HTML into the scene, not to be confused with the one mentioned above). A lot of time was spent on this. We enhanced the JS api as we move forward building the library - we added functionalities as we needed them. We then finalised by adding other components and properties such as support for model files and spin property, and bunch of other libraries such as bootstrap and jquery into the build.

1.6 - Similar Solutions

Aframe.io [8] - This is a similar solution to Wirtual in terms of giving the ability to develop 3D-VR applications using a html-like syntax. However it falls short on a couple of things. First of all, it is low level. You have to know about all advanced 3D concepts unlike Wirtual which handles everything automatically and makes it optional to manipulate such things. Next, it is not based on html-as-we-know-it. They created their own flavour of html, which will not provide much advantage to a person who relies on their knowledge of HTML. Most importantly, no built-in ability to render HTML into the scene (no CSS3D rendering support [6]). Last but not least, it does not support non-VR-ready desktop browsers [10].

React-VR [9] - Another good way of building VR apps, but it has the similar limitations with aframe.io example above. An advantage of this is that it has a great api to perform some tasks, but it is still pretty low level. It has some very basic solution to support css, but no support for rendering HTML (CSS3D rendering). Another downside of this library is, as its name suggests, it is only for React.

2 - Domain Analysis

2.1 - Target audience analysis

Since Wirtual is a library for developers to build product for end users, we have multiple target audiences. Our direct audience are developers. Developers will be using our library, our API's, and since we are an open source library, they will be in touch with us, either for reporting issues or contributing. Without developers, Wirtual project cannot live on, so perhaps the most important target audience for us are the developers. But developers will be building applications for end users using our library, which makes the end users our indirect target audience. If there are no users to use the products of our library, no developer will be using our library to produce anything. Therefor, end users are as important as developers. In this section, detailed analysis will be given on who is out target audience.

2.1.1 - Developers

Even though it is possible and easy, Wirtual is not intended for creating extreme graphics rich gaming / multimedia based content. It will not be the best option for developers who desires to build or has expertise on building such contents, with two exceptions mentioned in this section on 2.3 usage scenarios. Our intended target audience are the developers who has a working knowledge on web development, regardless of their experience level. Even with the most basic knowledge on HTML is adequate to be able to build a VR experience with Wirtual, and there is no limit for what could be done with it for increased level of advancement.

Our other developer target audience are the young developers. Programming is starting to take an important role in early education - an infographics by EurActive [11] reveals that already 15 EU countries have integrated programming in their school curriculum, 9 of them integrated / will integrate it into a primary school level. This means the world will be seeing more young minds who has the basic understanding of programming. Moreover, according to an article authored in 2012, 3D thinking can improve math and science skills, spatial thinkers are likely to be more interested in science and math [12]. With the programming knowledge gained from school level education, it is almost impossible for kids to program a 3D-VR applications amidst all the difficulties and complications mentioned in section 1.3.1. However using Wirtual's dead-simple HTML api, especially the 3D grid system, they can improve their minds thinking in terms of 3D space, while boosting their creativity and come up with amazing products.

2.1.2 - End-users

The end-user target audience of Wirtual is general public. We have designed this library so that anyone with a computer or a mobile phone can experience products built with Wirtual. It provides users either full enjoyable experience of VR, or what they might have experienced if they had the necessary equipment.

2.2 - Requirements specification

2.2.1 - Non-functional requirements

For the developers;

- **Ease of use** - Developers should be able to easily understand and use the library. It should eliminate the required knowledge to create 3D scenes, and overly simplify 3D environment coordinate system.
- **Customisability** - Developers should easily be able to customise the components provided by Wirtual according to their needs.
- **Maintainability** - Wirtual should provide a clean API to developers so that their code can be clean and maintainable.
- **Learnability** - The concepts of 3D-VR development should be overly simplified and eliminated from developers concern when possible, so developers could learn how to use the library and create their projects within matters of hours.

For the end-users;

- **Compatibility** - Wirtual should work with all Oculus devices, all Google Cardboard and Google Cardboard ready devices (HTC View on the Roadmap but not yet supported).
- **Browser support** - Following browsers should be able to run Wirtual: IE 10 or higher, Edge 12 or higher, Firefox 16 or higher, Chrome 36 or higher, Safari 9 or higher, Android 5-6.x WebView Chromium or higher, Chrome Android 55 or higher, Firefox Android 50 or higher, iOS Safari 9.2 or higher (bottleneck being CSS3D rendering as it is the newest and experimental feature). All these browsers are being supported by webvr polyfill.
- **Performance** - HTML codes compiles on the fly, any changes made to the DOM should be detected and the necessary components should be recompiled on the fly. So the compilation should be instant, there shouldn't be any lag.
- **Usability** - Users should be able to use the components only with intuition, without requiring a need for assistance.

2.2.2 - Functional requirements

For the developers;

- **Compile** - Providing an HTML api which developers can use to build the components, Wirtual should have the ability to scan and process those elements, build a DOM tree, map them to our 3D components, generate the output and expose it to the canvas element.
- **Re-compile** - Ability to detect changes to the DOM elements, then partially or fully re-compile and update the canvas in an efficient manner.

- **Exposure of JS Api** - Exposure of a Javascript api attached to the *window* object, which provides some functionality for developers to manipulate objects programmatically.
- **Exposure of Three** - The JS api should expose the instances of ThreeJS objects, giving advanced developers the ability to make use of all the possibilities of ThreeJS, the library powering Wirtual.
- **Adaptation** - Wirtual should have the ability to be used along with all modern JS frameworks / libraries, including Angular, React and Meteor.
- **HTML Rendering** - Developers should be able to write HTML code and be able to render it right into the 3D-VR scene. Wirtual should treat the rendered HTML just like any other 3D component, all the properties and rules that applies to common 3D objects and components should also apply to rendered HTML contents.

For the end-users;

- **Position tracking** - The web components will be distributed around the user, user should have the ability to look around using a VR headset, or dragging the cursor. Also Wirtual should make use of user's mobile phones gyroscopic sensors to allow them to look around by turning their phones around.
- **Ability to type** - Since users wearing a VR headset will not be able to see the actual keyboard, there should be a Virtual keyboard where they can do the typing by clicking.
- **Toggling modes** - Users should be able to easily toggle the cardboard mode on and off.

2.3 - Use Case Examples

One usage example that would fulfill the needs of one of the potential target audiences would be if Wirtual was used in schools, within educational context. After teaching students of young ages the basics of HTML, a small walkthrough of Wirtual could be given, and getting creative with world of VR world could be expected of them. One other expected use cases would be marketing and promoting content. There is a huge market for VR-related hardware and software nowadays. A good way to promote these commercial goods would be to make the marketing relevant and interesting - giving a demonstration of what to expect from that product with a demo webpage. Usage of Wirtual in this scenario would be ideal. Yet another example use case would be a real-estate agent's website. With the help of an 360° camera which is available in the markets, a real-estate agent could take pictures of a house, upload it and use Wirtual to allow visitors have a tour of the house in VR. He can also place arrow buttons for users to walk around the every room, he can attach text or images next to certain items for explanations, and he can get creative by implementing things like "click on the door to exit to the menu". One last usage example could be a social museum, where multiple users visit a museum and walk around the displayed items with the same logic of the real estate agent's website, but a small chat-box could be attached next to each display item and users examining the same item could have a chat with each other. This could easily be achieved by combining Wirtual with an isomorphic Javascript framework which utilises websockets, such as Meteor or Flux.

3 - Technical Details

3.1 - Dependencies

In this section, we will give information about all the libraries that powers Wirtual. We will list each of them, provide links to their git repositories, and reveal what is it that the library does for Wirtual.

3.1.1 - ThreeJS & Helpers

Three JS [4] is a cross-browser library to create 3D graphics that uses WebGL and provides an API to support all possible sets of features that has anything to do with 3D, including VR development. Even though its API is higher level that WebGL API itself, it is still quite low level and requires some level of expertise in 3D development and familiarity with its syntax. However, it is the pioneer of 3D-VR web development. It not only supports Wirtual, it also supports an array of libraries and projects including Aframe.io [8], Google's VR Chrome Experiments [13], MozVR [14] and many others. The number one dependency of Wirtual is ThreeJS, and the helper scripts that comes alongside its core. It's core supports the basic functionalities, and all the features it supports are distributed into modules that comes in different scripts files. Here are the ones Wirtual uses:

- **VRControls.js** [15] - Acquires positional information from connected VR devices and applies the transformations to a three.js camera object.
- **VREffect.js** [16] - Handles stereoscopic rendering for Google Cardboard devices.
- **ColladaLoader.js** [17] - Data loader for Collada (.dae) files, which is one of the supported 3D model formats along with OBJ + MTL.
- **OBJLoader.js** [18] - Data loader for Obj files.
- **MTLLoader.js** [19] - Data loader for MTL files.
- **CSS3DRenderer.js** [20] - Handles CSS 3D rendering. It acquires the properties given to its three.js instance, applies CSS 3D transformations to the targeted DOM elements and places the DOM elements as a layer over the canvas, makes it behave exactly like other Three.js elements.

3.1.2 - WebVR Polyfill & Manager

WebVR Polyfill [21] is, as the name suggests, a polyfill library that implements WebVR specs [22] in Javascript and makes it compatible with browsers that does not support WebVR. It is built and maintained by Google. This is the library that decides which VRDisplay to provide depending whether you browser, whether it is mobile or desktop. If you are on your mobile, it provides CardboardVRDisplay, which uses the devices gyroscopic sensors and also utilizes a technique called sensor fusion and pose protection [23] to provide orientation tracking. It is the

library that lets you switch between cardboard mode (CardboardVRDisplay) and desktop mode (MouseKeyboardVRDisplay). For this, it provides useful UI elements and UX interactions; it brings the gear icon to switch to the cardboard mode, back button to exit the cardboard mode, settings button to pick which version of Google Cardboard to browse with, and a useful screen where it requests you to switch your phone into landscape mode if you are on the portrait mode. WebVR Manager is just a tiny library that emits subscribable events on mode changes of the WebVR Polyfill library.

3.1.3 - Others

- **ES6 Promises** [24] - This tiny library is a polyfill for Javascript ES6 style promises for browsers which does not support Javascript ES6.
- **jQuery** [2] - An extremely famous and widely used Javascript library which intends to simplify client-side scripting and DOM manipulation. This comes built in to Wirtual, but may be reconsidered according to developer feedback. More on evaluation chapter.
- **Twitter Bootstrap** [1] - Another very famous front-end framework that contains built in styling and drop-in ready-to-use components. This also comes built in to Wirtual as well, and may be reconsidered according to developer feedback. More on evaluation chapter.
- **Animate.css** [3] - An easy and lightweight solution for drop-in CSS animations for DOM elements. This as well comes built in with Wirtual and may be reconsidered according to user evaluations.

3.2 - Build System

In this section; we will explain in detail how all the source codes of Wirtual gets compiled, merged with all the third party javascript and css libraries and bundled into one single javascript file, `*Wirtual.js*`. Each step of the automated build system is explained.

The command that starts all the build process is `npm start`. Npm commands are specified in the file `package.json` in the root directory. The `scripts:` property of the json file contains the bash scripts to trigger in response to the specified npm command. `Start` is a special command which could be invoked just by running `npm start`. Other parameters needs to be invoked by calling `npm run ...`. So `npm start` command triggers `npm run build`, which triggers the command `*rm -rf dist/; webpack --progress --colors --mode=build; grunt attach-vendors; grunt serve;*`. First part of this command (`rm -rf dist/`) removes the `dist/` directory, which contains the old build. Second part (`webpack --progress --colors --mode=build`) initiates webpack with a bunch of options. Third and the last part (`grunt attach-vendors; grunt serve;`) initiates specified grunt processes. So when this bash command is executed, first the `dist/` directory gets removed. Then, webpack starts. All the webpack related configurations could be found in `webpack.config.js` file in the root directory. Webpack is the system which compiles Wirtual source codes and merges into one file called `webpack-out.js`. It has a few responsibilities while doing this. First of all, Wirtual is built with Javascript ES6 standards, which is the final version of

javascript, and it is not supported by many of the browsers. So converting ES6 to ES5 is crucial for obtaining the most browser support. One of the most important task of Webpack is this. Other important task of it is module loading. Javascript ES6 is modular - it supports `import` statements as in Object Oriented languages like Java. But that is not the case with ES5. So Webpack's responsibility is to load all these js modules and merge them, but with an order that no module is used before it is defined. So given these, Webpack loads all modules, merges them in a certain order, converts it to ES5, and copies the output file into the `dist/tmp/` directory. Before this command is executes, neither the `dist/` directory, nor the `tmp/` directory in it exists, because they are deleted after the first command. Webpack creates those directories along the way. After webpack (second command) is done, there is a `dist/` directory, with a `tmp/` directory in it, with a `webpack-out.js` file inside of it. Next, grunt gets initialised. Grunt [25] is a task runner, which executes a set of tasks and used to automate the build system. Grunt configurations could be found in `Gruntfile.js` in the root directory. First grunt task that gets initialised is `attach-vendors` (third command). As the name suggests, it merges the vendors (dependencies) with each other and with the compiled version of Wirtual source code (`webpack-out.js` in the `/tmp` directory). However it doesn't just copies and pastes the codes of each library, it wraps them between `*(function(){*` and `*})();*` expressions, which turns each of the individual dependencies into an immediately invoked function expression (IIFE). This allows each library to get invoked right after it is created, keeps the execution context inside of a closure so it prevents internal variables from polluting the global namespace and avoids conflicts between each of the dependencies. Only what each of the dependencies (and Wirtual) exports to the `*window*` variable gets registered globally. So as explained, all dependencies, configuration files (config files will be explained further later in this section) and Wirtual source code gets merged inside an IIFE's and gets combined in a single file, `wirtual.js`, which then gets copied into the `dist/` directory. Last thing `attach-vendors` grunt task does is to remove the `/tmp` file as it is not needed anymore. This concludes the third command, and we are left with the `dist/` directory and the `wirtual.js` output file inside of it. Last command just starts a server from the `demo` directory and has nothing to do with the building of the library. It is to be used in the development process.

3.3 - Compiler

Before diving into the details of the compiler, here is a brief description of its responsibilities. It goes through all the DOM elements, gives every one of them a unique ID, saves some metadata of each element into the register, invokes necessary modules to be compiled, handles the data-flow while doing that, finally initiates the render loop and the heartbeat that watches for change. Here are some terminology used in the context of explaining the compiler, followed by the explanation itself:

3.3.1 - Terminology

Hashing - Hashing a DOM element is the process of extracting its HTML codes with everything in it (including all its children). Comparing hash values is how Wirtual detects a change in an element. However, there are some elements / properties that should not be included in the change detector, therefor in the hash. For instance, we don't want the HTML elements rendered with CSS3D renderer to be included into the change detection, because CSS3D renderer updates their DOM every millisecond (with the CSS 3D transformation coordinates required for it to adapt the 3D environment), and if we include them in the change detector, it will trigger a change and a recompile every heartbeat. Therefor hash function removes some properties / tags using regular expressions after extracting the HTML contents of an element. `wr-component`, `wr-render`, `data-wid` and HTML comments (<!-- -->) are examples of what hash function removes. It also removes spaces, tabs and new-line characters, because change in those are also not important and we don't want to waste memory keeping them. So the hash function returns the outerHTML of an element minus all the things that is removed. This is implemented in `src/compiler.js` with the name `hash(el)`.

Rehashing - When an element's DOM is subject to a change, it has to be hashed again with its new DOM, and the hash value in the register should be updated. Not only the element itself, all of its parents has to go through the same procedure. If this is not done, a change will be detected over and over again in each heartbeat, even after the necessary recompilations are done. So rehash method starts a recursive call starting from an element going up until the root, calling the hash function for the element and each parent. The changed elements unchanged children however does not have to be rehashed since it is not subject to a change. This is implemented in `src/compiler.js` with the name `rehash(wid)`.

Stamping - Stamping a DOM element is the process of generating a random ID that starts with the prefix `_w`, followed by a random 10 character string. The stamped elements gets a data attribute (attributes like `*data-spin="1"*` are called data attributes) called `wid`, so the `*data-wid="_w....."*` property could be observed in the html content of the stamped element. Moreover, stamping hashes the element after modifying its `data-wid` property. When stamp function gets the hash, it bundles the `wid` it generated with the hash of the element and a pointer to the elements DOM target within an object and saves it to the registry with generated `wid` being the key and this bundled object being the value. This is implemented in `src/compiler.js` with the name `stamp(el, options)`.

Marking - Marking is a tail recursive process which drives the stamping through the DOM elements. It is invoked with a starting point, and from the starting point, it goes down through all the children in a depth-first manner and calls the stamp function for each of them. It optionally could be called with a callback, and when it is done, it invokes the callback. This is implemented in `src/compiler.js` with the name `mark(wid, recursionLevel, options)`.

Compiling - Compile is a tail recursive method, it is similar to *marking* in the sense of starting from a node and going all the way down recursively. But the course of action that this method takes is different. It tries to match the class names / tag names of each element with the built-in phrases, and initiates the necessary objects accordingly. For instance, if the compiler finds the `wr-sphere` class name inside of a component, it will initiate the static `compile` method of the Sphere object (`*Sphere.compile(currentElemenet)*` - currentElement being the scanned element, which will be used by the Sphere class for compiling). As it could be understood from this explanation, actual compiling of components are being handled by their own modules, but the main compiler is the one that scans the DOM and initiates them. One other important function of this method is to handle the data flow. For instance, when the compiler finds the `*wr-level-**`, `*wr-depth-**` or `*wr-axis-**` classes, which are not meaningful by themselves but are intended to pass on coordinate data to all its children, the compiler appends the coordinate data to a payload object, and passes that object on to the children by making the next tail-recursive call with it. That's how child objects receives data from parents. This payload object is also saved into the register under the namespace of each children component, and gets synchronised after each recompilation, so when a child component (but not its parent) have to be recompiled, it can have access to previously assigned payloads coming from its parent. As an example to this data flow, if a `wr-sphere` div wrapped by a `wr-depth-100` div has to be compiled, `depth:100` property will be appended to the payload when the compile method for `wr-depth-100` is running. Next recursion for its child, `wr-sphere`, will be called with the payload object that has the `depth:100` property. So the child `wr-sphere` element will be compiled using `depth:100`, and `depth:100` will be saved into the registry for the namespace reserved for `wr-sphere` element for later use. If there should be a change to the DOM of the element `wr-depth-100`, say if it is updated to `wr-depth-200`, it will get recompiled, but this time passing `depth:200` instead of `depth:100` in the payload to its children (`wr-sphere`). And when this child `wr-sphere` is being compiled, it will first synchronise the new depth value of `200`, then recompile itself with it. And if there should be a change to the `wr-sphere` element but not to its parent `wr-depth-200`, the compile method will start directly from `wr-sphere`, and the parent will not be there to pass on `depth: 200` in the payload anymore (as the parent is not getting compiled). That is precisely why payloads are saved into the register, and in this case, the compiler of `wr-sphere` will restore the previously assigned payload which has the `depth: 200` property in it, and recompile itself with `depth` being 200. If the payload wasn't saved and synced with the registry, a slight change in a child's DOM (and not the parents) would result with the child element loosing the data that it is supposed to get from its parent, and in this case, `wr-sphere` would end up being recompiled in a different location than specified. This method is implemented in `src/compiler.js` with the name `compile(wid, payload, recursionLevel, callback)`.

Scanning - Also referred as 'the heartbeat', is the process of scanning the DOM for changes. It runs in every heartbeat, which is 50ms by default, starting from the root element. It may however be initiated from any other node. It first gets the data of the starting element from the registry. This data contains a pointer to the DOM target of the element, so it find the actual DOM element on the page following that link. Then, it runs the hash function for that DOM element,

and compares it with hash value that is fetched from the registry. If those two hash values match, that means there is no change in the element. But if they don't match, it means there is a change. So this is the point where the scanner decides whether it should start a compilation, step in (start a recursion from children) or halt. It has the strategy of starting a compilation from an element only when it finds an element that has a change and is a leaf node (which means the change is definitely within the element), an element that has a change and has unchanged children (which also means the change is definitely within the element), or an element that has a change in multiple children (as in most cases it will be more efficient to just recompile the parent rather than recursing until finding all changed children and recompiling each one of them). If however it finds an element that is changed and has only one changed child, it starts a recursion from that child instead of starting the recompilation from the parent. This strategy has two advantages: It allows selective recompiling - only the changed elements gets recompiled and others won't get affected; and also in every heartbeat, the scanner doesn't have to go through all the DOM nodes to make sure there is no change, so no-change case could be concluded with a single computation. This method is implemented in `src/compiler.js` with the name `scan(wid)`.

Render loop - The infinite loop which gives life to all the renderers, animations and frames. Also within this loop, the attached runnable methods gets executed. More detail on runnables will be given later in this section.

3.3.2 - Methodology

Compiler gets initiated within its constructor, and it gets instantiated right after the DOM is loaded in `index.js`. First thing it does is to look for the `wr-container` element, which is the root element that Wirtual lives in. Then, it *stamps* the root element and starts *marking* from it. When the marking is done, it starts the *compile* process, again starting from the root. The compile method goes through all the DOM three, instantiates the necessary modules, and when all elements on the page are done being compiled, it starts two things: the render loop and the heartbeat (scanner). More information on how the compiler works and stores data, enhanced by visual aspects, could be found here [26].

3.4 - Components

The component names mentioned in this section that starts with a dot (.) means that it is a class name. Tag names will be displayed like *<this></this>*. Data properties will be displayed like *data-x*.

.wr-container - Creates a scene

- data-brightness ('high', 'medium' *(default)*, 'low') - Determine the intensity of the light for the scene

- data-skybox (string) - Name of the folder that contains the skybox images, also the string that each image name has to start with, followed by an underscore ('_') and ('px', 'py', 'pz', 'nx', 'ny', 'nz') ($p^{positive}$ $n^{negative}$)
- data-skybox-format - Format of the images (ie. jpg, png)

.wr-light - Creates a point light attached to a small sphere.

.wr-sphere - Creates a sphere.

- data-size (integer) - Size of the sphere
- data-cover (string) - URL or path to an image, which will cover the sphere material
- data-color (string) - Hex value (or in defined words like 'red') will determine the color of the material if data-cover property does not exist.

.wr-emoji - Creates 3D emoji

- data-size (integer) - Size of the emoji
- data-smiley ('smile', 'tongue', 'mock') - Smiley type

.wr-text - Creates 3D text

- data-text (string) - Text to be displayed
- data-size (integer) - Font size
- data-depth (integer)
- data-curve-segments (integer)
- data-bevel-size (integer)
- data-bevel-thickness (integer)

.wr-model - Loads a 3D model file into the scene

- data-model (string) - Path to the model file
- data-model-format ('obj', 'dae') For obj files, it can also include the .mtl file in the same directory with the same name.

<wr-render></wr-render> - HTML within these tags will get rendered and added into the scene.

<wr-component></wr-component> - For now, acts exactly the same as wr-render tag. However the purpose of having this separately is that in the future, some component-specific properties will be attached to this.

Keyboard - This is a virtual keyboard that pops up when users focuses on a text input element. It's purpose is to give the ability to type to the users with VR headsets who cannot see the actual keyboard. When the JS api is initialised, it instantiates the Keyboard module, which binds the event listeners that watches for 'focus' events on input areas with a callback that fires itself.

3.5 - Api

3.5.1 - HTML Api

The HTML api consists of a set of class names and data properties that are used to manipulate the elements in the 3D scene without having to write code. The intention is convenience and

ease of use, however everything that could be achieved using the HTML api could be done with the JS api as well. Usually this api is used to manipulate the coordinate-related properties.

.wr-level-(integer) , .wr-depth-(integer) , .wr-axis-(integer degrees) - These properties are the handles of the 3D grid system Wirtual provides. These classes should be given to a container element, and the coordinate data will be passed on to the children so child elements will be compiled taking these coordinates into account. For all three of these values, 0 means the origin point, which is the eye of the user. To take full advantage of VR, user will be in the center and all components will surround the user. Level determines the vertical placement, negative values being below and positive values being above the eye of the user. Depth is the horizontal distance from the camera, elements with a positive depth value will be rendered in front of the user whereas elements with negative values will appear behind. Finally, axis represents the axis around the user, and it takes the value in degrees. For instance, 90 (or -270) will be on the right of the user, whereas -90 (270) be on the left.

data-spin="(double)" - Could be attached to any object, will make the attached object spin with the specified speed. It will attach a `spin: true` property on the instance of the component.
data-spin-direction="('left', 'right')" - Complementary to data-spin property, will determine the direction of the spin. Default is right.

Location manipulating - Location manipulating classes should be used within a button or a clickable element, instead of the object that is intended to be relocated. These classes vary between `*wr-move-** and `*wr-rotate-**`, which manipulates the positional and rotational values respectively. These helper classes consists of `*wr-move-up*`, `*wr-move-down*`, `*wr-move-left*`, `*wr-move-right*`, `*wr-move-near*`, `*wr-move-far*`, `*wr-rotate-xup*`, `*wr-rotate-xdown*`, `*wr-rotate-yup*`, `*wr-rotate-ydown*`, `*wr-rotate-zup*` and `*wr-rotate-zdown*`. For targeting the element to be moved / rotated, an HTML `id` property should be given to the desired element, and that id should be specified with the `*data-target="given_id"*` data property given to the button / clickable along with these classes. Moreover, `*data-delta="(double)"*` property could be used to determine the amount of change, which has the default value of 25. These classes could be used to move /rotate every 3D object in the scene.

3.5.2 - JS Api

What makes this JS api great is the exposure of the Three objects, giving developers the ability to do practically everything that Three.js is capable of. Wirtual api lives in the `window` object under the namespace `Wirtual`. Most functions in this api are for internal usage, and are specified with the generic naming convention of starting with an underscore (_).

Everything throughout this report that is mentioned by 'registry' or 'register' are stored within this api, under the object `Wirtual.dom`. It is basically a key - value store, with the stored elements forming a linked tree structure. Key is the `wid` property assigned by the compiler, and value is the object contains all the metadata. Since this store needs rapid access by the compiler to read

and write, the requirement was desired object to be located in instantaneous time, which is why the store was constructed in this manner. Some important properties that are stored for each object consists of `dTarget`, `vTarget`, `parent`, `children`, `hash` and `payload`. `dTarget` is a pointer to the DOM element, `vTarget` holds the instance of the element running inside the 3D scene, `parent` and `children` holds the `wid` of the parent and the children of the element. These two properties makes this store a linked tree. `hash` and `payload` properties are well explained in the compiler section. The `vTarget` holds the instance of the Wirtual component, the contents can vary between different Wirtual components, but all shares the `mainTarget` property which holds the Three.js instance. Through this instance, developers can achieve the full potential of the powerful library behind Wirtual.

Another thing the api holds is the instance of the virtual keyboard, under `Wirtual.keyboard`. This property contains a set of useful method, such as `show`, `hide`, `moveUp`, `moveDown` etc. that could be used to control the keyboard programmatically.

Some other methods that the JS api provides are as follows:

- **Wirtual.get()** - returns the instance of the API - usually intended for internal usage but could be used elsewhere
- **Wirtual.isDebug()** - Boolean, is Wirtual running in the debug mode.
- **Wirtual.getElementByID(wid)** - Takes the `wid` and returns the `mainTarget` (Three instance) of an element.
- **Wirtual.getRootElement()** - Returns the `wid` of the root element.
- **Wirtual.getScene()** - Returns the scene instance.
- **Wirtual.getRenderer()** - Returns the instance of the main renderer (not the CSS3D renderer).
- **Wirtual.getCamera()** - Returns the instance of camera.

Runnables - Runnables are a set of functions in closure that are invoked within the render loop, which are controlled by the compiler and stored within the api. These functions could be global, or could be a runnable of a specific component. The purpose of this is that when a component gets removed, all the runnable functions that powers it should also be removed along with it. When getting compiled, Wirtual objects attaches the necessary function which will power itself. But this feature is also exposed for usage of the developers.

- **Wirtual.attachRunnable(id, runnableName, runnable)** - This method attaches the runnable function that the argument `runnable` holds, with the runnable name, under the `id` (not to be confused with `wid`, this is the HTML id attribute given to the element) field that is provided by the argument. Once the `runnable` function is attached, it will be invoked on each render loop as long as the element with the `id` lives on.
- **Wirtual.attachGlobalRunnable(runnableName, runnable)** - This method also attaches a runnable function, but it attaches the function globally. It stores it by the `runnableName`, and `runnable` gets invoked every render loop.

- **Wirtual.detachGlobalRunnable(runnableName)** - This method detaches a runnable by the name, so it gets removed from the array of functions that are invoked in every render loop.

3.6 - Utilities & Configuration

Utilities class contains some useful methods to be used throughout the app. First to mention is `hasClass`, `addClass` and `removeClass` methods, which is used to manipulate the class names. `log` method is used for logging information on the console, mostly for debugging purposes. It logs out data only when the debug mode is enabled. It also styles the logs on the console, so that they could easily be spotted among the logs of other dependencies. `random` method returns an arbitrary string with specified length. `toDegrees` and `toRadians` are the methods used for conversion. `calculatePosition` method is the most notable, since this is the method which calculates the `x` `y` and `z` coordinates that objects should be in regarding the `depth`, `level` and `axis` values coming from Wirtual grid system. It also calculates the rotation value based on the axis, which ensures whatever axis the object should be placed on, that it would be always facing the origin point (ie. to the user).

Configuration files are located in the `config/` directory. There are three config files, `css-config.js`, `meta-config.js` and `webvr-config.js`. The `css-config.js` file contains the css codes that gets appended in a style tag to the html document that Wirtual is running in. It contains the base css properties that Wirtual has to have, as well as the CSS dependencies of Wirtual such as bootstrap and animate.css. The `meta-config.js` file similarly appends the `meta` properties that Wirtual needs to the html. Finally, the `webvr-config.js` file contains the configuration needed for the webvr-polyfill and webvr-manager dependencies.

4 - Evaluation

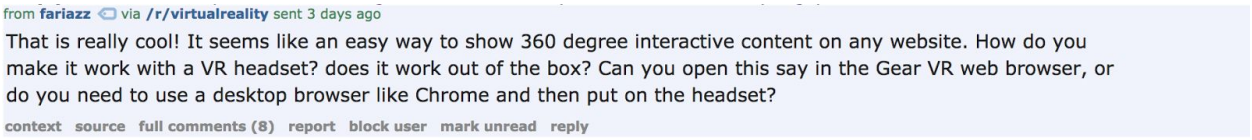
For evaluating Wirtual, we mostly consulted developers, since developer satisfaction is one of the most important aspect. We also asked some end users of their ideas on by showing them our demos, we will be talking about both in this section. The method we conducted for the evaluation was one to one interaction with each participant, because a method such as a user experience survey would be insufficient to give participants the idea since Wirtual the concepts we are working on is very abstract. The medium we used to reach out to the participants of our evaluation includes Reddit, Facebook, Slack and direct conversation.

4.1 - Developers

For consulting developers, we have talked with some of our classmates. We also wrote to quite a few developers in our inner circle through Facebook. We shared the demonstration video over

Reddit, and we asked the opinion of developers from Toptal (software company) slack group which has over more than 3000 developers. In the end, Wirtual was evaluated by over 30 - 40 developers. Most of the developer opinions were positive. Some of the responses were pointing out the similar libraries, which we mentioned in section 1.6.

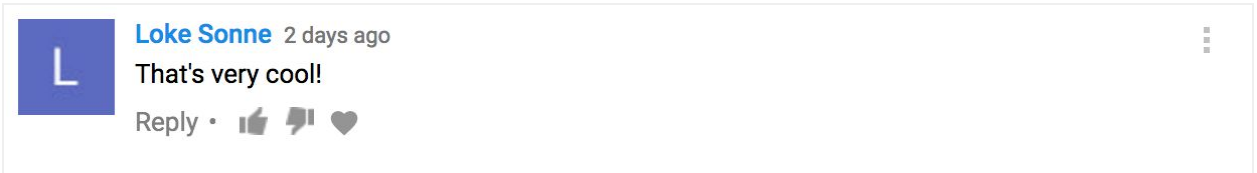
When told about the differences, the feedback was positive. What developers loved most is that the simplicity, and how Wirtual eliminates all the hassles of bootstrapping such a 3D-VR project.

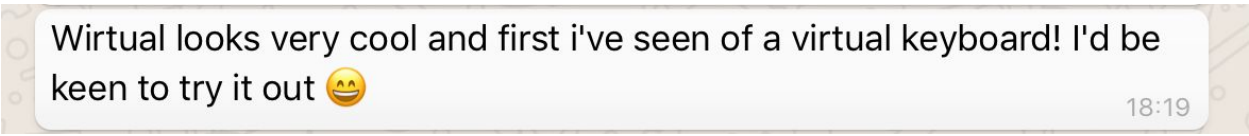


Some of the fellow developers were excited about it being a simple and easy way to display 360 degree interactive content. Some of the concerts that developers have was the inclusion of jQuery, Twitter Bootstrap and animate.css since those libraries could be added externally and does not have to be included in the source code. They do not have any effect on the functionality and the workings of Wirtual, so the were not sure about the necessity of embedding those into the source code. There was another negative comment that argues that the output file is too big (1.4 MB), which is a completely accurate statement. These two statements were somehow linked to each other, as those three libraries significantly increased the size of Wirtual. But as we also didn't want to omit the simplicity and completeness of this library, we ended up adding the future plan of enhancing the build system, make it produce two versions of the library, `wirtual.js` and `wirtual.lite.js`, with and without those libraries.

4.2 - End-users

If we were to sum up all the feedback we received from the end users in a single word, that word would be "cool".





General reaction from end users was being impressed. This was far from what they have seen in a browser before, they were excited to try it out. Usually the end users we have spoken to had a hard time imagining how it would work on a VR glass. This is something we might want to spend some time on, to explain end-users how this would work. Also the end users who had familiarity with VR was quite happy that this would work with their Google Cardboard compatible device. Unfortunately no user who owned an Oculus device was interviewed, however the expected reaction from them would be the similar if not same.

5 - Conclusion

Virtual reality in a web browser is quite unheard of. Most of the existing / ongoing effort on bringing VR to the web is currently being spent on graphically enhanced experiences; HTML components and basic level user interactions are not properly supported with existing technologies and are very experimental. Also as VR development is hard, the variety of products falls short. We realised that it doesn't have to be this way. So we created a library very easy to learn and supports about everything for both developers and users. For users, we provided support for a wide range of VR glass types, browsers and mobile phones. For developers, we crafted the library in such a way that it relies on DOM manipulation, in order to make it compatible with available modern frameworks. We made it compatible with frameworks that heavily relies on data bindings like Angular 1, reactive frameworks like Angular 2, frameworks that virtualizes the DOM like react, and isomorphic frameworks like Meteor and Flux. Along with all these support we gave, we included the built-in support for HTML rendering to be able to stick with our philosophy, something that no other available library supports by heart. With all we did with, we deeply hope Wirtual to boost up the amount of available VR projects, and make VR more accessible for all users. The setup instructions and the code base could be found here [27].

6 - References

[1] - http://getbootstrap.com
[2] - https://jquery.com
[3] - https://daneden.github.io/animate.css
[4] - https://threejs.org
[5] - https://hacks.mozilla.org/2016/05/exporting-an-indie-unity-game-to-webvr
[6] - https://aframe.io/docs/0.4.0/introduction/device-and-platform-support.html#support-for-vr-experiences
[7] - https://www.statista.com/statistics/426469/active-virtual-reality-users-worldwide
[8] - https://aframe.io
[9] - https://facebookincubator.github.io/react-vr
[10] - https://aframe.io/docs/0.4.0/introduction/device-and-platform-support.html
[11] - https://www.euractiv.com/section/digital/infographic/infographic-coding-at-school-how-do-eu-countries-compare

[12] - https://ww2.kqed.org/mindshift/2012/06/22/how-spatial-thinking-can-improve-math-and-science-skills
[13] - http://vr.chromeexperiments.com
[14] - https://mozvr.com
[15] - https://github.com/mrdoob/three.js/blob/master/examples/js/controls/VRControls.js
[16] - https://github.com/mrdoob/three.js/blob/master/examples/js/effects/VREffect.js
[17] - https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/ColladaLoader.js
[18] - https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/OBJLoader.js
[19] - https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/MTLLoader.js
[20] - https://github.com/mrdoob/three.js/blob/master/examples/js/renderers/CSS3DRenderer.js
[21] - https://github.com/googlevr/webvr-polyfill
[22] - https://w3c.github.io/webvr
[23] - http://smus.com/sensor-fusion-prediction-webvr
[24] - https://github.com/stefanpenner/es6-promise
[25] - http://gruntjs.com
[26] - http://oguzgelal.com/wp-content/uploads/2016/12/wirtual_compiler.pdf
[27] - https://github.com/oguzgelal/wirtual