Reference (Published version):
Marx, Robin; Vanhove, Sander; Vanmontfort, Wouter; Quax, Peter & Lamotte, Wim(2017) DOM2AFrame: Putting the Web back in WebVR. In: Proceedings on the International Conference on 3D Immersion 2017 (IC3D),p. 1-8

DOI: 10.1007/s11242-017-0940-y
Handle: http://hdl.handle.net/1942/25335

DOM2AFRAME: PUTTING THE WEB BACK IN WEBVR

Robin Marx[1]*, Sander Vanhove*[1]*, Wouter Vanmontfort*[2]*, Peter Quax*[1]*, Wim Lamotte*[1]

[1]UHasselt-tUL-imec, EDM, Hasselt, Belgium.{first.last}@uhasselt.be
[2]Androme, Diepenbeek, Belgium. wouter.vanmontfort@androme.be

ABSTRACT

As the Virtual Reality (VR) market continues to grow, so does the need for high-quality experiences. In order to unlock the wide pool of existing web-based content, common and more specialized web browsers allow users to visit existing web pages in VR. Additionally, the latest version of the WebVR [24] standard facilitates the integration of custom 3D/VR content in a web page. Sadly, both options consciously exclude the use of standard 2D web technology (such as HTML and CSS) in other common use cases, such as creating a highly interactive 2D UI for a 3D/VR game. Consequently, web developers wanting to use WebVR are required to learn an entirely new skill set to create VR experiences.

This work surveys and explores workaround options for rendering 2D HTML/CSS/JavaScript-based content in WebVR. We find that existing methods are often too slow to allow for a highly interactive experience. We introduce *DOM2AFrame*, a new framework that couples 2D page elements directly to their equivalent 3D counterparts (using the *A-Frame* library [2]) to allow for smooth updating, animation and user interaction at frame rates close to 60 FPS on modern hardware. Two case studies validate our approach and show its potential for rendering 2D web content in VR.

Index Terms— Virtual Reality, WebVR, rasterization, DOM

1. INTRODUCTION

Virtual Reality (VR) is on the rise [11]! The need for premium VR-enabled content has exploded in tandem with the availability to the general public of VR-capable hardware (e.g., headsets, controllers and personal computers). While this content is available in sufficient measure in the form of stereographic video and interactive game-related experiences, the rich ecosystem of (2D) web-based content has not yet been fully explored in the VR medium. This is a pity, as there are various interesting use cases (UCs) for using standard 2D web-based technologies such as HTML, CSS and JavaScript (JS) in VR. For example:

- **UC0** : Rendering and interacting with a full web page as-is
- **UC1** : Rendering a 2D User Interface (UI) or other 2D elements in a 3D experience (e.g., complex menu in a game, text reader)
- **UC2** : Rendering (parts of) a web page in a different way when viewed in VR (e.g., integrating 3D objects as part of an e-shop, auto-selecting stereographic video, horizontal instead of vertical placement)

Some recent developments try to unlock this large body of existing web content either in a special purpose VR browser (e.g., the Janus VR browser embeds web pages as part of a larger 3D world), or as separate VR rendering engine or view in an existing web browser (e.g., Google Chrome and Samsung Internet). Sadly, these browsers typically only allow the singular use case **UC0** and do not provide web developers with direct control over how their content is rendered in VR nor how the user can interact with it. This is problematic, as the browser's typical approach (simply displaying the site on a flat 3D surface (e.g., plane, cylinder)) is unlikely to produce a satisfactory user experience [16].

To support **UC1** and **UC2**, we might look towards the new WebVR standard [24]. WebVR provides an abstraction on top of the various VR hardware setups and enables developers to use the existing WebGL [10] rendering capabilities of the HTML5 `<canvas>` element [21] to easily create stereographic 3D experiences. Various VR input methods are made available through the JS-based Gamepad API [24]. WebVR is currently supported in both Google Chrome and Mozilla Firefox, with Apple's Safari recently joining the WebVR standardization effort [5].

In theory, web developers could thus use WebVR to render 2D web content in VR and support all three use cases. In practice however, modern browsers do not grant developers direct access to their rendering pipeline and also do not provide an API to render/rasterize typical web content onto a `<canvas>` element[1]. The main argument for not providing such a coupling or API has always been security and privacy related, as it could lead to abusers taking and saving "screenshots" of sensitive user data (e.g., an attacker could load an online banking page in an `<iframe>`, render it to a texture and send it to their server; while this is certainly a solvable problem, the possible solutions are not easy to implement in the rendering pipeline [20]).

For better or worse, web developers who want to create VR content are currently limited to the functionality provided by the 2D and 3D rendering contexts of `<canvas>` and WebGL, which means that most of the more powerful and established web technologies such as HTML, CSS and large parts of JS (e.g., input event listeners) are not directly usable in WebVR. This implies that web developers cannot use their established skill set to start creating VR content, instead requiring them to develop new skills (e.g., knowledge of 3D primitives, shaders), which are typically taught to game developers.

In this work, we first look at existing options to bypass these limitations and how to render full-featured 2D web content as part of interactive 3D and VR experiences (Section 2). We discover that

[1]Note that Mozilla Firefox does provide a similar API [1], but that it is only available from within browser extensions and plugins, not to normal web content.

developers have been using workarounds to render web content to 2D textures, then mapping them onto 3D primitives for use in VR. We find that these methods are far too slow to support interactive 3D WebVR experiences and are inflexible in their handling of user input. We contribute our own *DOM2AFrame* framework (Section 3), which does not first render to a 2D texture but instead couples each 2D web element to an equivalent 3D representation directly. Our approach provides fully interactive frame rates and more flexible input handling (with support for HTML forms), at the expense of the resulting render not being a pixel-perfect replica of the browser engine's result. We validate our approach in Section 4 with two case studies that show our support for **UC1** and **UC2** and, in part, **UC0**. Note that we look primarily at the rendering and input capturing aspects and consider techniques for interacting with (Web)VR content out of scope for this work.

2. RELATED WORK

2.1. Adjusting the native rendering pipeline

Rendering a web page is an iterative process (see Figure 1a). The browser engine starts from the raw HTML/CSS/JS strings and builds an internal representation: the Document Object Model (DOM[2]). This internal representation is then used in multiple steps, mainly Layouting and Styling. Layouting deals with how elements are positioned and sized (e.g., with CSS properties such as width, height, margin) while Styling changes the visual appearance of the elements (e.g., borders, color, font-weight). Once the Layout and Style are calculated, the browser can use this information to draw the rasterized representation of the web page via the GPU. When something changes on the web page, the browser only updates the relevant elements of its internal representation to prevent a full page re-draw for every update. It is thus straightforward for the browser to provide a VR-specific rendering viewport on top of this internal representation, since it mainly needs to change how the content is dispatched to the GPU and can reuse most of the existing implementation.

It is no wonder then that much of the previous work has focused on extending existing browser implementations to enable stereoscopic rendering and/or providing web developers with ways to access a VR rendering setup. In their seminal paper [6], Barsoum and Kuester use an external VR rendering engine and run an Internet Explorer web browser process in the background. An ActiveX plugin captures the web page's contents and streams the resulting bitmaps to the VR engine. Their results indicate low refresh rates of only 4-7 FPS. Jankowski and Decker [16] use a custom web browser to allow users to switch between a "normal" 2D HTML view and a 3D view on top of which the HTML content is overlayed in flat "windows". More recently, Wang et al. [26] have extended Webkit to allow for easy stereoscopic rendering of HTML content, introducing multiple APIs such as HTML-S3D, CSS-S3D and JS-S3D. Similarly, Lim et al. [17] propose several CSS extensions to allow for stereographic rendering and 3D manipulation, which they validate with an emulated rendering-engine[3].

Ultimately however, all these approaches require either a custom browser implementation/plugin and/or propose an extension to existing web standards, making them difficult to deploy in practice and in combination with modern APIs such as WebVR/WebGL. Additionally, most of the discussed methods focus primarily on supporting **UC0** and sometimes **UC2** (highlighting that enabling **UC2** is traditionally difficult without built-in browser support), but rarely **UC1**.

2.2. Rendering to a 2D texture

As discussed in the previous sections, rendering web content in VR without changing browser source code is difficult because a developer cannot *directly* access the browser's internal rendering pipeline through JS or any other means[4]. However, there are some workarounds that obtain a very similar result by using the available 2D/3D rendering contexts of the `<canvas>` element. These methods first render (large) groups of DOM elements to a 2D image/texture, either by using the internal pipeline in a roundabout way via SVG images (Section 2.2.1) or by re-implementing the browser's Styling logic (Section 2.2.2). This texture is then mapped onto a WebGL 3D (quad) primitive for use in VR.

Two figures aptly summarize the next two Sections: Figure 1 shows the various pipeline details, while Figure 2 clarifies how each method maps web elements onto multiple textures.

2.2.1. rasterizeHTML and html2three

Using the special `<foreignObject>` tag of the XML-based Scalable Vector Graphics (SVG) image standard, HTML and CSS content can be embedded in an SVG image element. The `<canvas>` 2D drawing API then allows to draw/rasterize the SVG image, causing the browser to internally render the SVG in largely the same way as it would with normal HTML and CSS content, using the default browser rendering pipeline (see Figure 1b).

However, once again due to security and privacy considerations, various web features are not directly supported. For example, JS code execution/events are disabled inside the SVG and external JS cannot access the resulting (sandboxed) SVG DOM copy. This means that interaction with form elements (e.g., `<input>`, `<textarea>`) is not supported. Additionally, most external resources (e.g., images, CSS stylesheets, fonts) cannot be directly included and must first be downloaded outside of the SVG and then inlined as base64 encoded strings. There are many edge cases involved in this process and browsers do not support all possible HTML and CSS features (e.g,. hyperlinks, see Figure 3b ❶). Various open source implementations of this methodology have arisen, for example *rasterizeHTML* [9].

The experimental *html2three* library [18] uses *rasterizeHTML* internally to render a web page in a WebVR context. A limited amount of top-level `<div>` containers are rendered as separate quads (one texture/quad per container) (see Figure 2a and 2b, nr. 1 and 4 are individual parent containers). Each container element is

[2]In reality, what we are referring to as the DOM is actually a combination of DOM, CSS Object Model (CSSOM) and executable JS code and couplings (e.g. event handlers). We use the summarizing term DOM here for brevity and clarity.

[3]A W3C proposal adding Stereographic 3D to CSS fizzled in 2012 [22].

[4]Note that while there are methods for overlaying DOM content on top of a 3D `<canvas>`, these do not support stereographic views.

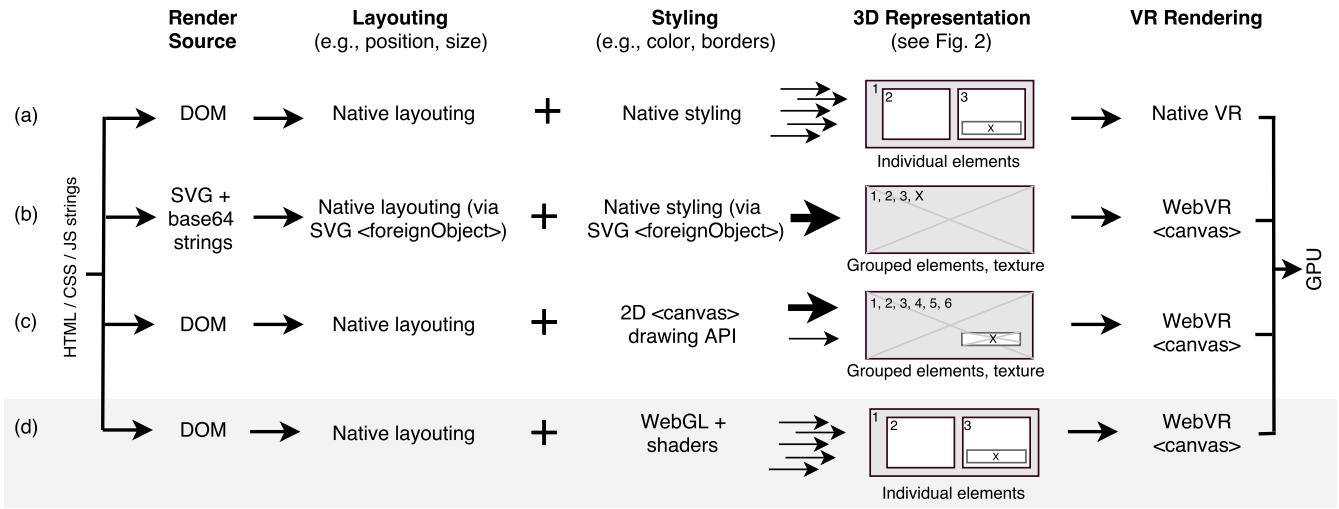


Fig. 1: Rendering pipeline: (a) native browser, (b) *html2three* (*rasterizeHTML*), (c) *HTML GL* (*html2canvas*) and (d) *DOM2AFrame*.

seen as a "sub page" and can be separately re-rendered. Interaction is provided by capturing pointer event coordinates in the UV texture coordinate space and then looking up the corresponding DOM element through the standard `document.elementFromPoint` API, which can find DOM elements through document space coordinates. Any resulting changes to the original content are then captured through the JS Mutation event system (see Section 3) and can result in a full re-render of the triggering content's parent `<div>` container by creating a new `<foreignObject>` SVG.

2.2.2. *html2canvas and HTML GL*

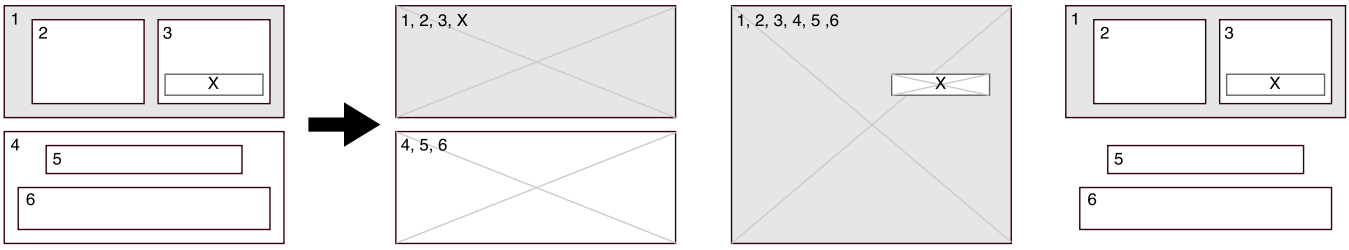
The core idea behind the *html2canvas* library [19] is to completely re-implement the Styling and rendering/drawing logic in JS. To this end, *html2canvas* uses the 2D drawing `<canvas>` APIs directly to render custom versions of the DOM elements into larger 2D textures, see Figure 1c. The native browser's Layouting engine is still used to determine relative element positioning and size, enabling complex CSS positioning such as floating and flex grids (see Section 3 for more details). While this approach is flexible, it requires large amounts of custom drawing code in the Styling step to obtain results that are visually similar to those of the native browser. That being said, *html2canvas* appears to be fairly mature and up-to-date, as its visual fidelity level was found to surpass that of *rasterizeHTML* (compare Figures 3c and 3b).

The proof-of-concept *HTML GL* library [12] adds 3D/VR capabilities to *html2canvas*. Like *html2three*, *HTML GL* will render groups of DOM elements into a limited amount of individual textures, but it uses a different heuristic to determine how to group the DOM content. Where *html2three* partitions elements into several larger parent containers, *HTML GL* instead tries to improve performance by creating a separate 2D texture for a few select elements, namely those which have a custom CSS 3D transform set (e.g., using CSS translation or rotation). If and when those transforms change, *HTML GL* no longer has to update the larger texture; it can instead directly apply the transform to the corresponding 3D quad by constructing the correct 3D transformation matrix and skip the texture generation. For example, in Figure 2c element X has a CSS 3D transform set and is rendered apart from the other elements as a separate texture.

Like *html2three*, *HTML GL* handles pointer-based input through `document.elementFromPoint`.

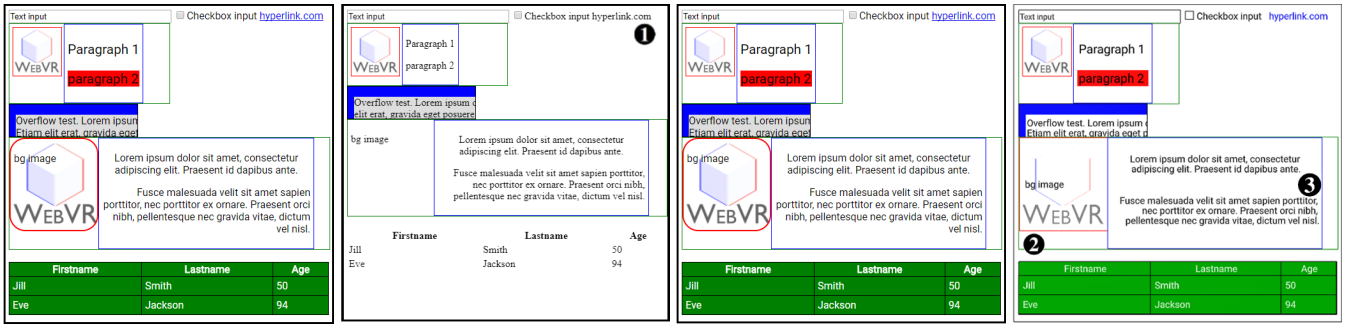
2.3. Discussion

Existing academic work proposes extensive and optimized methods for rendering DOM content in VR, but often requires changes to the web browser's source code, making it difficult to use in practice. In contrast, rendering to a 2D texture is a directly applicable workaround and can clearly lead to good overall visual replicas of the browser's behaviour (see the comparison in Figure 3), but it also has severe downsides. Firstly, the method is inflexible in terms of input capturing (e.g., input is no longer directly received on an individual 2D element, form element support is limited). Secondly, re-rendering a large 2D texture is often not fast enough to provide interactive frame rates when there are many changes to the DOM (see Section 5). A naive optimization would be to simply render a separate 2D texture for each individual DOM element to improve the update performance. In practice however, the JS API overhead for creating and managing these textures quickly adds up and this "optimization" turns out to hurt performance when applied in anything but a very coarse way (i.e., more than about 5 - 10 individual textures in our tests). This is why the discussed implementations choose different middle roads (see Figure 2): *html2three*'s approach (b) works well in cases with a few largely independent container elements, while *HTML GL* (c) will perform best if there are multiple moving parts spread around the page. This is possibly also one of the reasons why neither of the two 3D libraries supports the `<video>` tag, even though this can be done by texture-mapping the internal video framebuffer.



(a) *native* (b) *html2three (rasterizeHTML)* (c) *HTML GL (html2canvas)* (d) *DOM2AFrame*

Fig. 2: Mapping DOM to 3D: (a) individual source elements, (b) render to texture (two textures total), (c) render to texture with optimization for X (two textures total), (d) full direct mapping (no textures, seven dynamic elements of which one (nr. 4) is not rendered)



(a) *native* (b) *html2three* (*rasterizeHTML*) (c) *HTML GL* (*html2canvas*) (d) *DOM2AFrame*

Fig. 3: Visual fidelity: (a) the browser's 2D ground truth. (b), (c) and (d) same page rendered using methods from Sections 2 and 3.

3. DOM2AFrame

Our contribution, *DOM2AFrame*, aims to provide a web standards compatible solution for rendering 2D DOM in 3D/VR, with high update performance and flexible input support. While our approach is conceptually similar to *html2canvas* in that it employs custom Styling and rendering logic, it differs in that instead of using the 2D `<canvas>` APIs, we use the 3D WebGL capabilities to remove the intermediate step of rendering to a 2D texture (see Figure 1d and Figure 2d). We create and maintain a direct mapping between the DOM elements and their 3D counterparts, making it easy to only amend the 3D representation for the DOM elements that have changed after an update. As such, we achieve a dual performance boost by a) getting rid of the 2D texture overhead and b) only updating what is needed. We receive (pointer-based) user input directly on the correct 3D element (via raycasting/"picking") and can pass it on to the original DOM element through our maintained mapping. The main downside of our method is that it is difficult to achieve a high degree of visual fidelity when compared to the browser's ground truth for complex Styling setups (see Section 3.3).

A proof-of-concept implementation of *DOM2AFrame*, along with all discussed test pages, case studies and several movies are available at `https://webvr.edm.uhasselt.be`

3.1. DOM to A-Frame mapping

In practice, most of *DOM2AFrame*'s features do not use the low-level WebGL capabilities directly, but rather rely on Mozilla's higher-level *A-Frame* library [2, 23], which in turn uses the lower-level Three.js library. *A-Frame* allows users to build up their 3D VR scenes in HTML-alike markup, see for example the bottom part of Listing 1. This markup is then parsed by JS at runtime and used to create 3D elements, providing web developers with a familiar way to declaratively compose VR scenes [8]. Note that most of the technical challenges discussed in this work are inherent to our overall approach and not due to the choice of *A-Frame* as a base framework. Indeed, similar results can be achieved by using for example Three.js, the ReactVR framework [4] or a variant on the X3DOM approach [7].

Listing 1: *DOM2AFrame* example (excluding positioning/sizing)

```
<img src="logo.png" />
<p style="background-color: red; color: blue;">
    ↪ Lorem ipsum dolor</p>

   Is transformed by DOM2AFrame into:

<a-asset id="abc123" type="img" src="logo.png" />
<a-image src="#abc123" />
<a-entity>
   <a-plane color="#FF0000" />
   <a-text color="#0000FF" value="Lorem ipsum
       ↪ dolor" />
</a-entity>
```

However, despite *A-Frame*'s HTML-alike markup, it only supports custom elements that usually have a direct 3D equivalent (e.g., `<a-box>`, `<a-plane>`) and not the standard HTML elements (e.g., `<div>`, `<p>`), nor does it support direct CSS manipulation

of these custom elements. So, while *A-Frame*'s approach looks and feels like HTML-based development, it is still not directly usable for developers lacking previous 3D experience.

To overcome this issue, *DOM2AFrame* creates automated custom mappings from a variety of HTML/CSS elements onto equivalent *A-Frame* objects and WebGL shaders. It recursively loops through the children of a chosen root element and creates a corresponding *A-Frame* element for each. For example, Listing 1 shows how a `<p>` and `<img>` element can be transformed into equivalent *A-Frame* representations for rendering. For many HTML elements that mainly serve as a container for other content (e.g., `<div>`, `<table>`), a simple `<a-plane>` is a sufficient analogue. Elements that display text are composed out of both an `<a-plane>` and an `<a-text>` element, as the latter doesn't directly support a background color or borders. Other, more specialized HTML elements (e.g., `<select>`) can require more complex setups but, in general, all the HTML elements necessary for our case studies (see Section 4) could be composed out of simple *A-Frame* components.

Once the mapping between DOM and 3D elements is built, *DOM2AFrame* uses the browser's native Layouting calculations to position and size the elements in 3D, courtesy of the `getBoundingClientRect` method (available on all DOM elements). This method returns the coordinates and size of the DOM element's bounding box relative to the document's top left corner, after the browser has positioned all the elements based on their CSS properties. This approach means *DOM2AFrame* supports any type of complex CSS positioning out of the box (as do the other methods discussed in Section 2, which use the same API), but also that the original 2D page and its DOM must remain active (though it can be hidden).

In contrast to the Layouting step however, the browser doesn't provide easy access to its native Styling/drawing engine and we need to re-implement most of the Styling logic ourselves. The browser's internal CSS representation (the CSSOM) can be retrieved through the `window.getComputedStyle` interface, which returns the interpreted values for all CSS properties. Luckily, for many of these properties, there are direct *A-Frame* analogues that can be used (e.g., text and background color, background image, text alignment) and that are often implemented directly in fragment shaders. Other properties are less straightforward. For example, *A-Frame* elements by default do not have the option to set a border, while CSS offers many flexible possibilities in that regard. *DOM2AFrame* therefore currently only supports borders for rectangular elements by creating new thin triangle strips/lines on the edges of the `<a-plane>`. Another example of a difficult CSS aspect is that of overflow (i.e., hiding content where it is larger than its parent's bounding rectangle). *html2canvas* achieves this by using `<canvas>`'s 2D `clip` method, which discards pixels outside of the clipping area during rasterization. This feat is much harder to accomplish in 3D however. Our current solution is to use four individual orthogonal clipping planes (one per edge of the rectangular element) and discard pixels outside these planes in the element's fragment shader. Additionally, *DOM2AFrame* also includes support for various non-standard HTML attributes/elements and CSS properties. These allow the developer to more easily manipulate the appearance of their HTML content in 3D/VR or to create a web page that looks and behaves differently depending on whether it is viewed in VR or not (to maximize the re-use of source code).

Lastly, handling pointer-based input events (e.g., mouse, VR controllers) is relatively easy. *A-Frame* has provisions for a plethora of cursor-based interaction concepts, including "gaze clicking" and uses raycasting to find the correct element. *DOM2AFrame* then intercepts these *A-Frame* events and transforms them into equivalent DOM events (e.g., click, hover, mouseout). It can then dispatch the DOM event to the correct 2D element, which then triggers the registered existing JS event handlers and keeps support for event bubbling intact.

3.2. Updates and animations

Given the *A-Frame* representation of the DOM and the mapping between corresponding 2D and 3D elements, we can efficiently reflect any DOM changes by updating the correct *A-Frame* attributes or creating/deleting *A-Frame* elements. The challenge here lies in how to actually detect those DOM changes via JavaScript.

The browser provides a standard `MutationObserver` API [3], which can be used to observe a DOM element (and its children) and which generates `Mutation` events when something changes (e.g., CSS properties are changed, a new CSS classname is assigned, a child element is created). While this setup captures most mutations, there are a number of important edge cases. For example, CSS animations and transitions do not trigger the `MutationObserver` and as such require separate event listeners (e.g., on animationstart and animationend) and update loop logic (via `requestAnimationFrame`). Additionally, if an element changes size/position due to a CSS property change, this will generate a `Mutation` event for that element exclusively; other elements that might have been repositioned/resized due to this change need to be checked independently. Lastly, if a CSS property is not set directly on an object but rather by using a general CSS selector (e.g., `*{color:#FFF;}`), this will also not trigger a `Mutation` event. Currently, we know of no way to observe this specific situation in all possible ways it can occur and instead ask the developers to manually call an update method on all affected elements.

3.3. Limitations

The main challenges of our approach lie in the trade-off between satisfactory performance and reaching a visual fidelty that matches the browser's 2D rendering. Firstly, most of the core problems originate from the fact that several CSS style properties do not have a direct or simple counterpart in WebGL and its shading language. For example, we would need to implement/generate complex custom shaders/border meshes to support rounded corners (see Figure 3d ❷). Additionally, supporting content overflow hiding through clipping planes in the fragment shader works, but it does not scale well, as both the parent element and all of its children need to perform the clipping in their shaders individually. An interesting optimization could be to change the element's base mesh definition at creation time to match the clipping setup.

Secondly, font rendering is approached differently in 2D and 3D. Without direct access to the browser's font rendering engine (and thus also the .ttf, .woff and other web standard font format

Fig. 4: The Hologram case study

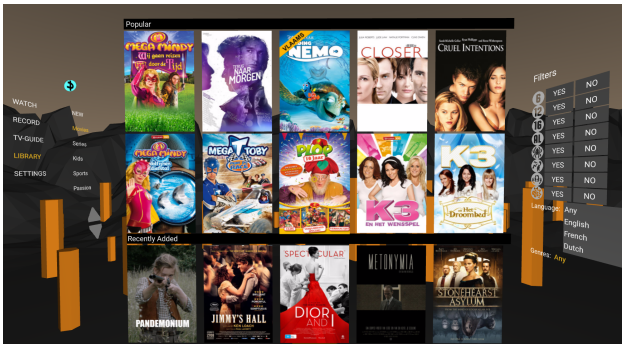


Fig. 5: The VodLib case study (All movie image copyrights belong to their respective owners).

implementations), *DOM2AFrame* resorts to other methods for 3D text. *A-Frame* supports bitmap-based fonts but also the more advanced *(multi-channel) signed distance field* (MSDF) font rendering method [14]. In practice, the differences between the *A-Frame* MSDF implementation and the CSS font conventions are so large, it is difficult to find a consistent/robust mapping between the two. Our implementation works well in some cases but shows large discrepancies in others (see Figure 3d ❸). It would require fine-tuning on a per-font/per-case basis to get a pixel-perfect replica of the browser's text rendering behaviour in *A-Frame*.

Finally, working directly with the WebGL backend means that *DOM2AFrame*'s performance is more likely to be GPU-bound (as opposed to CPU-bound). Especially on lower-end/mobile hardware this could mean our method becomes slower than more CPU sensitive methods (see Section 2.2.1 and 2.2.2). Additionally, using complex fragment shaders means being weary of fill rate limitations (i.e., how many pixels can be drawn per frame). For example, in Figure 2d, if element 1 is completely drawn and shaded before elements 2 and 3, time is wasted on computing eventually invisible pixels. Even though most GPUs and rendering logic already take this into account (i.e., by depth-sorting elements first), not rendering invisible DOM elements helped performance considerably in our tests (e.g., elements such as nr. 4 in Figure 2d, which have no background color/border and serve only as positioning/sizing containers for other contentful elements).

4. CASE STUDIES

In order to validate *DOM2AFrame*, we implement two custom case studies. We take care to integrate HTML/CSS functionality that is not/less supported by related work (e.g., form input, video, animations) to highlight the benefits of our approach.

4.1. The Hologram

This first case study primarily aims to show that *DOM2AFrame* can be used to implement use case 1 (**UC1**): Rendering a 2D User Interface (UI) or other 2D elements in a 3D experience. The demo simulates a mainframe user console in a futuristic setting, see Figure 4. It includes an "accordion" menu to the right (where non-active menu items slide out with a CSS transition as the active item takes up all available space, implemented using CSS overflow), a video display that can toggle between flat video and overlayed stereographic video in the middle and a form that can be used to select animations on a 3D "hologram" (top). Console commands can be executed at the bottom, causing them to be added to the third accordion pane on the right. This demo is also fully functional in the non-VR view: the 3D hologram model is rendered in a flat `<canvas>` and only the non-stereoscopic flat video is played. In this way, it is also a validation of **UC2**.

4.2. Video-on-Demand library

Our second case study aims to show that *DOM2AFrame* can be used to implement use case 2 (**UC2**): Rendering parts of the web page in a different way when viewed in VR. This demo (see Figure 5) shows a "Video-on-Demand library" (VodLib), similar to such sites as Netflix, where users have an overview of video content and they can browse and choose an appropriate (stereographic/360°) movie to watch. The filter options on the right, which would typically be checkboxes on a normal website, are instead automatically presented as push buttons in VR for easier user interaction. Similarly, a normally horizontal drop down menu is replaced with two adjacent vertical lists (left). The two side menus are also rotated around their vertical axis to achieve a better impression of presence in the 3D world. Note that this demo is complementary to that of Hugo Hedelin [15], who also implements a VR-based VodLib, but focuses on the user interaction aspects and implements the 2D UI directly in *A-Frame*.

By way of a third demo, we attempted to use *DOM2AFrame* for use case 0 (**UC0**): Render a full web page as-is. Our implementation was able to correctly render large parts of a complex existing VodLib web page, but was unable to deal with some complex CSS concepts, such as "pseudo-elements/classes" (e.g., `:hover`, `::before`) and complex background image setups. Currently, it is often difficult or impossible to manipulate or interpret these CSS rules from inside JS. We expect this situation to change in the future with more powerful CSS APIs [25].

Aspect	Method			
	Non-VR Browser *(Google Chrome)*	**rasterizeHTML** *(html2three)*	**html2canvas** *(html-gl)*	**A-Frame** *(DOM2AFrame)*
(a) Allows UC1 & UC2		✓	✓	✓
(b) Pixel perfect DOM equivalent	full	largely	largely	limited
(c) Input support	native	coordinate mapping	coordinate mapping	raycasting/picking
(d) Form support	full	very limited	limited	custom, extensive
(e) Video playback	✓			✓
(f) Single element frame duration (ms)	16.6	90	370	16.9
(g) Full testpage frame duration (ms)	16.9	275	380	18.1
(h) Interactive framerates (+ animations)	✓			✓

Table 1: Comparison of different methods of rendering DOM content to 3D/VR. For frame (ms) measurements, lower is better.

5. EVALUATION

In order to further evaluate the performance characteristics and feature-completeness of the discussed methods, we used a custom test page which includes a variety of different HTML elements (e.g, `<img>`, `<video>`), CSS layouting (e.g., floating, overflow) and styling (e.g., rounded borders, background color) directives, triggerable CSS/JS animations, DOM manipulations and input event handlers. Part of this test page can be seen in Figure 3.

The reported results in Table 1 were obtained in Google Chrome v59, on an Intel i7 7700k desktop computer with 16GB of RAM running Windows 10, using an NVidia GTX 1070. Tests in Mozilla Firefox v54 and on other (slower) desktop hardware found quantitatively similar results. We consciously decided to generate our results on high-performance desktop hardware because we feel most mobile hardware is not yet powerful enough to run full VR experiences and that this might skew our results, as the hardware rather than the software becomes the bottleneck. Consequently, the reported "ground truth" results are for the native non-VR Google Chrome browser, as the VR-specific viewport of Google Chrome is currently only available on the Google Daydream headset (which uses a mobile phone for its processing). We speculate that the native desktop browser will also be able to reach this optimal performance in a hypothetical VR view because it is able to reach optimal results in the non-VR view.

Items a - e of Table 1 highlight the feature-completeness of the discussed approaches. While the native browser seems the winner in all options, remember that it only allows for browsing full web pages in VR (**UC0**), not for embedding interfaces (**UC1**) or adjusting how parts of the page are displayed (**UC2**) (see Section 1). While especially the more mature *html2canvas* library outshines our approach in terms of visual adherence to the browser's ground truth, *DOM2AFrame* is more flexible in terms of complex input handling and dynamic aspects such as video playback.

The performance aspects of the evaluated methods are shown in items f - h of Table 1. We define interactive frame rates as being between 30 and 60 frames per second (FPS)[5], corresponding to 33.3ms - 16.6ms individual frame durations respectively. We report the average frame durations, recorded over a period of approximately 30s, in which a variety of automated, JS-triggered animations and DOM manipulations take place. The measurements were repeated 10 times for each test case and we report the lowest measured average (optimal result). We can see that *rasterizeHTML* and *html2canvas* are limited to 3.6 FPS and 2.6 FPS respectively when animating the full test page (Table 1g). Even when constantly re-rendering a page with only a single `<p>` element (Table 1f) (considered as an "optimal" case), we see a maximum of 11.11 FPS for *rasterizeHTML*, while *html2canvas* does not show significant improvement, adding to our hypothesis that its low performance is not due to drawing overhead but rather texture creation and management (both pages generate only a single texture for *html2canvas*). In contrast, *DOM2AFrame* achieves an average 55 FPS for the full page and 59 FPS for a single element. As such, only our contribution and the native browser are able to reach interactive frame rates.

6. CONCLUSION

In this work, we have discussed the challenges in rendering standard 2D HTML/CSS web content in a (Web)VR environment for a variety of use cases. We have found that existing, web-standards compliant methods, which first render the web content to a 2D texture before moving to 3D, are often too slow to support truly interactive VR experiences. We contribute *DOM2AFrame*, a framework which creates a direct mapping between DOM elements and their equivalent 3D counterparts, enabling high update rates and extended interactivity. While there remain unresolved issues in our approach that are left for future work, we have validated our method on two concrete case studies to show that *DOM2AFrame* can render a large subset of web content in VR with high visual fidelity.

It is our hope that *DOM2AFrame* and similar approaches can help web developers to more easily create WebVR-compatible content by relying on their known and trusted web technologies. We also hope that through this work we can encourage browser vendors to reconsider providing direct access to the browser's internal rendering and rasterization layers, as with the advent of WebGL/WebVR, new use cases have arisen that would benefit from native rasterization APIs.

[5]Note that while 30 FPS is the minimum advised by the WebVR authors, various other work recommends using 60 to 90 FPS or higher [13]. The browsers used in our tests allowed a maximum frame rate of 60 FPS (vsync).

7. ACKNOWLEDGEMENTS

This work is part of the imec ICON PRO-FLOW project. Robin Marx is a SB PhD fellow at FWO, Research Foundation - Flanders, project number 1S02717N. Thanks to messrs Wijnants, Michiels and our anonymous reviewers for their help.

8. REFERENCES

[1] drawWindow. https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/drawWindow, 2017.

[2] Mozilla A-Frame. https://aframe.io, July 2017.

[3] MutationObserver. https://developer.mozilla.org/en/docs/Web/API/MutationObserver, July 2017.

[4] React VR. https://facebook.github.io/react-vr/, July 2017.

[5] WebVR WG participants. https://www.w3.org/community/webvr/participants, July 2017.

[6] Emad Barsoum and Falko Kuester. WebVR: an interactive Web browser for virtual environments. In *Electronic Imaging 2005*, pages 540–547, 2005.

[7] Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. X3DOM: A DOM-based HTML5/X3D Integration Model. In *Proceedings of the 14th International Conference on 3D Web Technology (Web3D)*, pages 127–135, 2009.

[8] PW Butcher and Panagiotis D Ritsos. Building Immersive Data Visualizations for the Web. In *Proceedings of International Conference on Cyberworlds (CW17)*, 2017.

[9] Christoph Burgmer. rasterizeHTML. https://cburgmer.github.io/rasterizeHTML.js, June 2017.

[10] Dean Jackson, Jeff Gilbert. WebGL. https://www.khronos.org/registry/webgl/specs/latest/2.0/, June 2017.

[11] Deloitte. VR: A Billion Dollar Niche. https://www2.deloitte.com/global/en/pages/technology-media-and-telecommunications/articles/tmt-pred16-media-virtual-reality-billion-dollar-niche.html, 2016.

[12] Denis Radin. HTML GL. https://github.com/PixelsCommander/HTML-GL/tree/v2, June 2017.

[13] Sebastian Friston and Anthony Steed. Measuring latency in virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, pages 616–625, 2014.

[14] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, pages 9–18, 2007.

[15] Hugo Hedelin. Design and evaluation of a user interface for a WebVR TV platform developed with A-Frame. Dissertation. http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-134948, 2017.

[16] Jacek Jankowski and Stefan Decker. A dual-mode user interface for accessing 3D content on the world wide web. In *Proceedings of the 21st ACM international conference on World Wide Web*, 2012.

[17] Soon-Bum Lim, Hee-Jin Lee, Wen-Yan Jin, and Seung-Min Shim. CSS3 extensions for setting web content in a 3D view volume and its stereoscopic 3D display. *Computer Standards & Interfaces*, 50, 2017.

[18] Marcio L. Teixeira. html2three. https://github.com/marciot/html2three, March 2017.

[19] Niklas von Hertzen. html2canvas. https://html2canvas.hertzen.com/, January 2016.

[20] Robert O'Callahan. Risks Of Exposing Web Page Pixel Data To Web Apps. http://robert.ocallahan.org/2011/09/risks-of-exposing-web-page-pixel-data.html, September 2011.

[21] Robin Berjon. HTML5 Canvas. https://www.w3.org/TR/html5/scripting-1.html, October 2014.

[22] Soonbo Han, Dong-Young Lee. Extensions for Stereoscopic 3D support. https://www.w3.org/2011/webtv/3dweb/3dweb_proposal_121130.html, November 2012.

[23] Srushtika Neelakantam, Tanay Pant. *Learning Web-based Virtual Reality*. Apress, 2017.

[24] Vladimir Vukicevic. WebVR. https://w3c.github.io/webvr/spec/latest/, July 2017.

[25] W3C. Houdini. http://dev.w3.org/houdini/, July 2017.

[26] Wenmin Wang, Shengfu Dong, Ronggang Wang, Qinshui Cheng, Jianlong Zhang, and Zhongxin Liu. Stereoscopic 3D Web: From idea to implementation. In *Information Science, Electronics and Electrical Engineering (ISEEE)*, 2014.