Creating IoT-ready XR-WebApps with Unity3D

Philipp Fleck
Graz University of Technology
philipp.fleck@icg.tugraz.at

Dieter Schmalstieg
Graz University of Technology
schmalstieg@tugraz.at

Clemens Arth
AR4 Gmbh
clemens@ar4.io

ABSTRACT

The rise of IoT-ready devices is supported through well-established web concepts for communication and analytics, but interaction yet remains in the world of web browsers and screen-based 2D interaction during times of tablet and smartphone popularity. Transforming IoT interaction concepts into 3D for future exploitation with head-worn XR devices is a difficult task due to the lack of support and continued disengagement of game engines used in XR development.

In this work, we present an approach to overcome this limitation, tightly including web technology into a 3D game engine. Our work leverages the versatility of web concepts to create immersive and scalable web applications in XR, without the need for deep-tech know-how about XR concepts or tiring customization work. We describe the methodology and tools in detail and provide some exemplary XR applications.

CCS CONCEPTS

- Software and its engineering → Software creation and management;
- Computer systems organization → Embedded and cyber-physical systems;
- Sensor networks;
- Human-centered computing → Mixed / augmented reality; Virtual reality; Graphical user interfaces.

KEYWORDS

XR, IoT, 3D Engines, Web browser, Web app

1 INTRODUCTION

Fast growing Internet of Things (IoT) and industrial sensor systems become more and more difficult to interact, explore and maintain. Furthermore, information is more likely to be cross-referenced among various domains (sensors, machinery, factories and others) to enable supportive applications like in-situ data discovery, training, maintenance and repair. Within an application the user has often to explore and consume data from different sources e.g., inspecting faulty machinery in terms of both physical components and virtual sensor data or documentation.

Collecting (sensor) data for analysis has a long tradition where all sorts of applications, libraries and technologies have been implemented, tested and strongly used. Nowadays and for the near future most systems are build around web technologies, especially visualizations. Tools like Grafana (grafana.com), Prometheus (prometheus.io) and Graphite (graphiteapp.org) utilize libraries like Chartjs (chartjs.org) and D3js (d3js.org) for rendering and styling. Common to all of them is their runtime in the web browser, mostly on a JavaScript (JS) foundation, and their highly compatible nature and portability to almost any hardware.

Emerging technologies like eXtended Reality (XR)1 bring a new level of immersive interaction and information consumption to the user, not only in industries but also to consumers. They are perfect to convey virtual information to users in a comprehensive and accessible way beyond the borders of desktop monitors. Nevertheless creating XR applications leveraging all this existing technology and modern IoT concepts is not a straight forward task. Besides the understanding of web and IoT technologies, it requires expertise in f.e., 3D Engines, Software Architecture and XR concepts.

In our work we propose techniques and concepts on how to create XR web-applications to extend existing IT-infra-structure for straight forward integration of heterogeneous virtual data. The major aim of this work is to relieve some burden from developers, app- and content-creators. We show how to allow a Unity3D based system to utilize web-technologies and existing content to create WebApps (electron alike applications, capable of running within the browser, but also directly on the system) with 3D-Engine support, still capable of running in the web browser.

In the remainder of this paper, we mainly focus on AR-applications for Head Mounted Displays (HMDs, aka smart-glasses) like the Microsoft Hololens. However, most of the principals apply to handhelds, VR-Headsets and XR in general as well. Our major contributions are the following:

- An easy way for creating WebApps with Unity3D and browser support
- Accessing and Visualize (sensor) data inside Unity3D
- Automatically creating visualizations for given data
- Utilizing existing infrastructure for AR applications
- Exploiting full web browser support in Unity3D
- Fast prototyping by hot loading

2 RELATED WORK

An exhaustive overview about visualization, IoT/web and XR technologies is beyond the scope of this section. It is very difficult to

---

Footnotes:

1 The term eXtended Reality (XR) refers to the collectivity of Augmented Reality (AR), Mixed Reality (MR) and Virtual Reality (VR) technologies.

2 https://www.electronjs.org
group approaches based on scope and to bridge the gap between those domains either. In order to still provide a comprehensive overview about relevant concepts from different domains, in the following we briefly discuss several approaches related to our work.

2.1 Web and Communication Concepts

Recent web technologies allow to build cross-platform applications, completely running in a web browser or a web enabled system. Many have shown how to build website running on mobiles with HTML5, CSS and JS support [David 2012, GitHub Inc. 2018; Toman et al. 2019]. Web libraries and frameworks like D3js [Bouali et al. 2016; David and J M Tauro 2015] allow real-time data handling and the implementation of nice and interactive visualizations. A core advantage is rapid prototyping (often bypassing the build stage of an application and utilizing in-browser evaluation like hot-loading) as presented by Stringer et al. [Stringer et al. 2005].

The drive to automate tedious tasks and to make decisions based on data was always a drive for new technologies and algorithms. SCADA [Boyer 2010; VanderZee et al. 2015] have a long history reaching back to the early 1970’s. This sensors managing systems and standards are capable of data flow handling and control on device level. Opening standards allowed cross manufacturer communication, leading to new independent technologies. NodeRed [Lekić and Gardašević 2018; Scicari et al. 2019] is a quite young open-source tool for sensor management, automation and more. Sensors and actuators ranging from buildings-infrastructure up to production factory machine control can be easily maintained and operated through such a system. With the rise of IoT, communication paradigms like MQ-Telemetry-Transport (MQTT, previously Message Queing) have been developed and deployed at small and large scales, often producing big amounts of data [Chanthakit and Marjani et al. 2017].

2.2 3D Content in Web Browsers

Web browsers are basically 2D interfaces, but early approaches to include 3D date back to the late 1990s. Standards like WebGL (https://webvr.info), threejs (https://threejs.org) and others support 3D content in a 2D browser nowadays. Often, device support on a low level is required and, therefore, it is not plausible to run full 3D applications inside a browser on an AR-enabled device. Software projects like Dom2Frame by Marx et al. [Marx et al. 2017] tackle such problems by rendering the DOM into an interactive texture. Speiginer et al. [Speiginer and MacIntyre 2019] described how to annotate existing 2D web content for immersive representation in 3D by manipulating an existing sample application. Furthermore, existing Dom-To-Texture techniques (Dom2Texture, Html2Canvas3) were shown to be sufficient for an immersive and fluid experience.

Use cases like city exploration and navigation benefit most from AR support in the browser to provide immersive experiences. The Argon web browser by MacIntyre et al. [MacIntyre et al. 2011; Speiginer et al. 2015] implements, amongs other things, KARML, an extension of Google’s KML4 (a geographical markup language) for geo-located information and enables the use exiting web technologies. More recently, Speiginer et al. [Speiginer and MacIntyre 2019] depict the layered reality model describing an independent software architecture to implement so called layers of reality. Those realities are decoupled and independently loadable similar to separated applications. Key proposed concept is write once, adapt everywhere and react continually.

2.3 Data Exploration in XR

XR lives by 3D virtual content. Often this refers to 3D models for either entertainment or productivity purposes in the first place (i.e., virtual twins). However, this paradigm even more applies to information of simpler nature, i.e., 2D data, potentially single numbers or sensor readings, but at a large scale with a plurality of data meanings and temporal and spatial dependencies. This is particularly important for professional applications in industry.

Grubert et al. [Grubert et al. 2017] describe the taxonomy and importance of sensing users context and adapting the AR system accordingly based on given constraints. This is coined context-aware XR with in-situ data visualization and exploration. Funk et al. [Funk et al. 2017] presented and evaluated an assisting system to support workers with repeating tasks, leveraging camera-projector systems. In order to generalize user interfaces for connected devices, Nichols et al. [Nichols et al. 2004] use parameterized templates to specify things like look and feel, button placement and size, situational behavior. Digital hardware with physical controls (e.g., music composition hardware, industrial machines) have haptic feedback but lack visual feedback. ControlAR by Jones et al. [Jones and Berthaut 2016] remixes graphical interfaces and creates an immersive visualization for such controls. Building immersive data visualizations allows to exploration and understanding in new dimensions. Butcher and Ritos [Butcher and Ritos 2017] build and evaluate VR visualizations utilizing existing web frameworks. By the example of spatial bar-chart an interactive prototype is showcased based on WebVr, A-Frame, WebGL and threejs. Bach et al. [Bach et al. 2018] examine effects of interactions in immersive AR applications by comparing desktop-AR vs. tablet-AR vs. HMD-AR (Hololens). Their study reveals that smart glasses show high potential and usability in the future, but undergo hardware limitations like small field-of-view, battery-lifetime and computational performance. Sicat et al. [Sicat et al. 2019] created an immersive analytics framework based on Unity3D with DXR and a grammar inspired by Vega5. Overcoming complex low level programming with descriptive grammar opens the door for more applications. Re-useable templates and graphical marks allow for rapid prototyping. Due to its underlying structure DXR is performance hungry and does not scale well, which is unimportant for in-lab use but crucial for in-field use.

The increased complexity of connected digital systems raised the need of understanding such networks, particularly important in maintenance of real assemblies and data analytics. Ivy [Ens et al. 2017] provides an interactive visualization of such networks, specifically depicting data-sources, data-sinks and data-flows. XR devices enable the application of such spatially situated visual programming tools for real-time usage.

---

1https://html2canvas.hertzen.com
2https://developers.google.com/kml
3https://vega.github.io/vega/
2.4 Bringing Web and 3D Engines together

Immersive XR applications are 3D heavy by nature, not mandatory content wise, but spatially e.g., location of the camera (Device) relative to the scene (Environment). 3D engines like Unity3D or Unreal (unrealengine.com) are therefore heavily used for XR development. However, as 3D engines originate from game development in the first place, they almost entirely lack the support of browser technology or well known 2D web content handling.

At the time of writing, the Hololens is the AR device of choice running Windows Holographic (at subset of Windows 10). A common approach is to deploy a Unity3D application as Universal Windows Platform (UWP) App in 3D mode. The XAML-2D mode is also supported and often a better fit for non-immersive 3D-Applications [Freeman and Freeman 2010; James et al. 2015]. Liu et al. [Liu et al. 2018] performed a hardware evaluation of the Hololens proofing good enough tracking and visualization capabilities for immersive AR applications.

The bottom line is that despite the availability of plenty of tools and even suitable hardware nowadays, getting the best of both worlds, i.e., 2D web and real 3D immersion, at the same time at scale is prohibitive. The only known approach tightly integrating 2D web technology into XR in game engines was recently proposed by Fleck et al. [Fleck et al. 2020]. They demonstrate an application for remote collaboration based on presented workflows and concepts, however, only briefly referring to the relevance of 2D web technology. In contrast, in the remainder of this paper we elaborate in detail on how to bridge different domains of data retrieval, data-visualization and application-design to enable rapid prototyping and, most importantly, webapp-style application creation through the integration of 2D web technology into a 3D game engine.

3 WORKFLOW AND TECHNICAL DETAILS

Current WebApps use a wide range of well established techniques for development and deployment e.g., Grunt, Gulp, WebPack and others\(^6\). Best practices are already commonly used across industries and developers. Such WebApps are the foundation of our daily web-life and tools for almost everything exist. In this work we try to leverage this potential within immersive XR applications. 3D engines like often implement their own User-Interface-Description-Languages for User-Interface (UI) creation and lack web-support. Such approaches have a high learning curve and require and profound skillset. Recently, Unity3D applies web principles to its UIs by introducing UIElements\(^7\), to ease the creation of UIs in a web-style manner.

Our approach, tries to fill the gap of missing web-support in Unity3D, allowing for fast prototyping and strongly decoupled UIs from compiled and published applications. The created WebApp can run in the web browser without Unity3D, allowing high code reuse since we can extend an existing (2D) WebApp to support AR on the Hololens. We utilize and extend PowerUI\(^8\) (a lightweight and fast JS interpreter and HTML, CSS renderer) to create meaningful and easy maintainable 3D applications with web support based on Unity3D. Other assets like WebView for Android\(^9\) often lack support UWP, especially the Hololens. We choose Unity3D over other 3D engines because of its wide hardware support starting from mobiles phones and desktop computers up to AR and VR headsets.

HTML, CSS and JS are used to stylize and implement applications, similar to best practices for web development. An example application capable to run on desktop, mobiles or smart glasses with Unity3D support is depicted in Figure 1, Figure 2 and Algorithm 1. The fairly simple application has one click-able button to open a web-site using JS for the click-event. Furthermore in-browser rendering and in-app rendering look alike. We have the following options to load the WebApp inside Unity3D:

- compiled storage: StreamingAssets or Resources
- file system: `file path or persistent data` depending on the Platform (slightly different for iOS, Android, UWP)
- URL like a web browser

---

\(^{6}\)gruntjs.com, gulpjs.com, webpack.js.org


\(^{8}\)github.com/Kulestar/powerui, github.com/philfleck/powerui

\(^{9}\)https://assetstore.unity.com/packages/tools/gui/3d-webview-for-android-137030
Figure 3: Common MQTT message flow: (blue) depicts three sensors publishing data to the topics \( A \) and \( B \). (purple) shows the MQTT-broker, the core unit, handling message distribution and quality-of-service (e.g., message can be temporally cached to serve the last message to new consumers). (orange) indicates the clients (aka consumers or subscribers) retrieving messages (in real-time) for their subscriptions.

All methods lead to the same result, where loading from file holds a small performance gain on the Hololens for complex UIs e.g., lots of nested <div> tags.

Tools like electron, a cross-platform desktop framework for WebApps, are used to create web-based desktop apps providing bindings from native (python/C++) to web (JS) and vice versa. With PowerUI and Unity3D we bind from C# to JS and back. On mobiles, the problem with web-content is not directly obvious, since in mobile AR we avoid interactive user-interfaces in 3D space because of the 2D touchscreen. A common approach is to overlay a native web-view over the 3D surface in which Unity3D is rendering.

Whereas, HMDs allow for easier interaction in 3D due to the immersive nature. Therefore web-content can not just be overlaid, but has to be rendered within the 3D engine. Since the Hololens is one of the last reaming HMDs after the fall of ODG\(^{10}\) and Daqri\(^{11}\) parts of our implementation are tailor towards UWP. We are limited to the restrictions of Windows Holographic, a highly locked down version of Windows10, disallowing easy service interaction e.g., off-screen-rendering in a different application and forcing to complex workarounds. Therefore we implement different techniques to allow web-support in Unity3D.

Usually applications serve a dedicated purpose and mostly the interaction is within the context of the application, within a set of windows (e.g., while interacting with MS Word one stays within MS Word). Immersive applications in XR allow interactions within the 3D world and its objects. Consequently, we need features surpassing a web browser, but offered by 3D engines f.e., supporting special hardware (HMDs). Being able to access features like raycasting from an 2D web browser interface enables us to depict any interaction paradigm within UIs: 2D ↔ 2D, 2D ↔ 3D and 3D ↔ 3D e.g., an action in 3D causing an actuation/reaction in 2D.

3.1 MQTT and IoT connectivity

Interconnected services are a crucial part when it comes to information availability, information discovery and infrastructure complexity. With MQTT on-site, service discovery and services maintenance has become much simpler. MQTT is a lightweight and scale-able, TCP based standard with many applications to it. Centered around a message-broker, clients can register to topics to receive and publish data as shown in Figure 3. Key concepts are asynchronous notifications and lightweight messaging. Most common use-cases are pinging (event notification) and small-data delivery. Compared to RESTful over HTTP, MQTT allows polling free implementations, which is especially beneficial for IoT (sensor-telemetry) data, due to the performance savings. However, low power HMDs benefit most, resulting in highly responsible applications. Since MQTT is not designed for large payloads, we can utilize the strength of both (RESTful and MQTT) to ship a lot of data by notification (see Figure 5). Furthermore we can decouple devices from applications and drastically increase code reuse-ability and code-complexity.

Before going into details of such implementations, we have to clarify how such an infrastructure (backend) should look like. Non invasive extensions to existing infrastructure are a set necessity to existing and stable running infrastructures. Usually services, endpoints and (IoT)devices which should be discover-able, are often implement as HTTP based API e.g., RESTful, GraphQL\(^{12}\) and similar. An application receives MQTT-topics over RESTful and directly connects to selected sensors over MQTT for real-time data streaming. Further API endpoints provide historic data for data analysis. Such backend designs, as shown in Figure 4, allow us to implement meaningful, easy-to-write and easy-to-use WebApps.
function MyWebGameObject () {
    try {
        var UE = importNamespace("UnityEngine");
        var myGo = new UE.GameObject("mywebgo");
        var newPos = new UE.Vector3(1,-2,3);
        myGo.transform.position = newPos;
        } catch (err) { /*"Not supported in Browser!"*/
    }
}

Algorithm 2: Calling Unity3D functions from JS

### 3.2 Client-side WebApp in Unity3D

The current state of the art in industries uses dedicated, mostly custom written, WebApps for data-visualization, inspection and retrieval. Single page views (e.g., showing current production numbers) are often visualized in hallways, offices or on certain locations close to the production area within a factory. Having the infrastructure and services in place, we can now explore and access such data. Using lightweight template engines like json2html\(^3\) we create data-driven, JSON-based templates for rich and interactive web-views (visualizations). Figure 6 and Algorithm 3 show how to create such an interactive visualization based on a JSON-transform. Transforms are transformed to HTML when feed to json2html. JSON-lists and other data-structures (e.g., lists of sensor-values) can be automatically processed by using markups within transforms (e.g., \( \langle !MYVALUE \rangle \)). Libraries like jQuery\(^4\) can be used accordingly. Unfortunately, PowerUI’s JS-interpreter is not feature-complete and therefore lacks full support of some special JS-libraries like ChartJs, D3js. In the latter of the section we provide a solution to this shortcoming.

### 3.3 Interoperability

Running a 3D Engines as app foundation allows us leverage specific functionality like raycasting, texture rendering and access to tracking data. Since PowerUI exposes and maps accessible C# functionality to JS we can implement independent code for such applications. The limitations of that approach are runtime-reflections and runtime-object-extensions. Furthermore, we can fully access Unity3D’s API and directly modify GameObjects, spawn Prefabs and more. As shown in Algorithm 2, we can enclose the interop-code in a try-catch-block maintaining browser compatibility, otherwise a Function not found is thrown. A common workflow is to use application related content within JS and only execute necessary calls to C#.

**Callbacks.** are a strong tool to implement asynchronous behavior, especially when designing interactive web-views. In C# callbacks are usually implemented via EventHandler or lambda functions, therefore it is easy use them. However, in JS we can use functions or lambda functions inside JS, but have to take care when receiving a callback from C#. Within the WebView we have three possibilities:

1. passing a defined function by name with a fixed set of parameters
2. passing a lambda function
3. passing a function by value (e.g., its name)

Since the C#-layer can access any present DOM, we can set which DOM will receive the callback, allowing us to perform inter-DOM-callbacks e.g., having a loose main-view to control a separate graph-view.

### 3.4 Hot loading content and functionality

From application maintenance perspective the concept of updating without shipping new app-versions (new compilations) is highly interesting. Since the whole application logic and UIs can be loaded anytime it opens a lot of possibilities. Updating and bug-fixing app-behavior becomes very easy. The proposed infrastructure with MQTT in place, allows for notify-and-pull concepts. Patches can easily be distributed, even at run-time. Furthermore, specialized implementation like new protocols, data-stream interpretation can be added at any time. Even sensors can add functionality to the application after connection (e.g., adding new virtual controls). An implementation and user based evaluation of the hot loading concept was presented by Fleck et al. [Fleck et al. 2020]. Similar to a web browser content is streamed or loaded at run-time from web (or other) sources.

### 3.5 Spatial (Web)UIs in 3D

Hence we apply Dom-to-Texture techniques, we can map a texture of the rendered web-view on any object or structure in 3D (e.g., on a 2D-Plane). Usually we have one DOM per texture, which allows us to render any web-content next to each other. As mentioned in section 3.3 we can cross-call among DOMs to easily implement menu or control interfaces. Placing 2D web-content in 3D we can adjust the depth layer separation. A bigger displacement will cause HTML-layers to spatially separate in the normal direction of the surface. Furthermore, known concepts like billboard and spatially registration are applicable to enable immersive experiences. Raycasting through the 3D scene is used to enable point-and-click interaction e.g., by gaze direction or hand movement. Actual clicks are registered by transforming the hitpoits texture-coordinates to DOM-coordinates. Lastly, we have to determine the size of our rendered view, since we are mapping pixels to spatial measurements:

\[
\frac{p}{ppw} = m
\]

Dividing the amount of horizontal pixels \( p \) by the spatial resolution (pixels per world unit) \( ppw \) we obtained its size in 3D in world units \( m \) (a width of 1000px and a spatial resolution of 1000ppw results in a 1m wide area).

### 3.6 XAML based WebView (web browser support)

In certain cases, full web browser support is required e.g., online video playback, handling specialized libraries or similar uses which go beyond the capabilities of PowerUI. The current version of Windows on the Hololens makes it hard to run software like headless render daemons when running a Unity3D UWP application in 3D mode. Usually UWP applications are paused when another one is started in Windows Holographic. Instead of deploying from Unity3D to an 3D app, we deploy it as a 2D XAML application and exploit the ability to switch between 2D and 3D mode. In a preparation step we add an additional XAML-page containing an...
Replacing RESTful polling with MQTT signaling: One method to get updates over a RESTful API without websockets is to do polling (e.g., for updating arbitrary status information). Enforcing two bad things: unnecessary connections and bad code. Using MQTT for signalling and to transfer the RESTful API endpoint via a message, we can still update huge amounts of data when needed, freeing client side resources (RAM usage, CPU time and bandwidth).

WebView, an extra library holding the WebView’s instance and code to auto-switch to 3D. Using the saved instance, we can access the WebView from Unity3D while running the 3D mode (D3D). Utilizing the screenshot function of the WebView (CapturePreviewToStreamAsync) we can snapshot and render the loaded web-page to a texture. Figure 7 depicts a cube running the live-stream of a local tv-station. The tricky part is getting interactions to work properly. Hence only a small set of the WebView is exposed, we use JS code injection to perform clicking. By knowing the WebView’s XAML resolution and the texture resolution, we can raycast the texture (applied on a 3D Object) and transform the hit into WebView-coordinates representing the DOM, as shown in Algorithm 4. To actually click, we execute an JS-eval call and try to find an element at given location. Once found, Click() is executed on the element to perform the click (see Algorithm 5). The biggest bottleneck when running on the Hololens is a slow texture update (approx running at 15fps) related to the mid-range hardware and the rather high idle-load of the CPU.

4 CONCLUSION AND OUTLOOK

In this work we presented workflows and concepts on creating IoT-ready WebApps. Furthermore, we showed pitfalls and provided solutions on how to get real web-support in Unity3D powered applications for XR. By allowing rapid prototyping through hot loading, we can provide a more web-a-like feeling on application creation. Applications can be fully written in web-languages like HTML, CSS and JS lowering the entry bar for developers and content creators. The presented extensions to existing infrastructures bring IoT connectivity and easy integration for XR apps. However, showing real-time patch- and update-able WebApps powered by Unity3D drastically eases application maintenance and reduces
code complexity. Additionally, existing techniques for deployment like Jenkins (https://jenkins.io) can be leveraged. Our concepts of inter-system-communication allow for high scale-ability and sustainability improving the overall quality of such services. Problems with lesser supported web-technologies in 3D engines will vanish in future, since a new trend started with the introduction of Unity3D’s UIElements. Furthermore, hardware makers have to put more thoughts in how their hard is used and allow to loosen some of the hard restrictions (e.g., configurable application sandboxing).

ACKNOWLEDGMENTS

This work was supported by FFG grant 859208.

REFERENCES


