DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Working in VR: a Web-based approach combining 2D and 3D Interfaces

Lorenzo Russo da Costa Auer

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Working in VR: a Web-based approach combining 2D and 3D Interfaces

Arbeiten in VR: ein web-basierter Ansatz zur Kombination von 2D und 3D Schnittstellen

Author:	Lorenzo Russo da Costa Auer
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Sandro Weber
Submission Date:	15.08.2020

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.08.2020 Lorenzo Russo da Costa Auer

Acknowledgments

Foremost, I would like to thank my supervisor, Prof. Gudrun Klinker, for giving me the opportunity to write my bachelor's thesis at her research group Forschungsgruppe Augmented Reality (FAR).

I also have to thank my advisor, Sandro Weber for his constant support, helpful advice and quick reactions.

Furthermore I want to thank Kilian Popp for his time and helpful advise.

Lastly I want to thank my family, my girlfriend and Kiba for all the support they gave me during my time studying and writing this thesis.

Abstract

Working with virtual reality is gaining a lot of popularity. Its capabilities of taking the user into a immersive 3D world, lead to it being used to simulate real world scenarios and observe or interact with them. With VR now also being available for browsers, a lot of new applications can be realized. One of them being, to introduce website interaction inside a VR environment. This thesis describes the options of implementing such a feature and the hurdles that were encountered, while trying to do so. The project has been realized with the ubi-interact networking framework, Vue.js UI components and the three.js 3D rendering framework. In the following text basic 3D rendering terminologies and methods are explained. Followed by the introduction of modern web technologies. The Three.js framework and and similar projects like mozilla hubs are mentioned and described, along with other related topics. Leading to the actual procedure of implementing the desired feature. Furthermore the hurdles like the same-origin policy applied by browsers, or the current limitations to WebXR are discussed and possible solutions to overcome them are introduced. Finally future plans and ideas, on how this project can be further improved, are presented.

Contents

Glossary

attack vector method an attacker can gain unauthorized access to a network or computer. 10, 35

backend describes the data layer of an application, the part responsible for all the logic behind the scenes, that is not visible for the user, for example data transaction. It interacts with the frontend. 1, 12, 19, 21–23, 29, 31

CORS Cross-Origin-Resource-Sharing: a mechanism, that allows browsers to execute cross-origin requests, which normally would be forbidden by the same-origin policy. 10

DOM Document Object Model: specification for programming interface, that represents HTML and XML files in hierarchical form. 9, 17, 28, 34, 35, 42

ECMAScript also known as ES; general-purpose programming language; standartized by Ecma International; core of JavaScript. 15

frontend describes the presentation layer of an application, containing the graphical user interface and everything the user can see. It interacts with the backend. 1, 19, 22

glb the binary version of gLTF, which stores the textures directly, instead of referencing them as external images. 31, 33

gLTF file format for storing 3D scenes and models using the JSON standard. 1, 31, 33, 40

HTML HyperText Markup Language: textbase language to describe electronic documents containing texts, hyperlinks , images and other content. 9–11, 14, 17, 22, 26, 29, 33, 35, 37

HTTP HyperText Transfer Protocol: stateless protocol, developed for transfering data in a network connecting multiple computers. 10, 12, 14, 15, 34

HTTPS HyperText Transfer Protocol Secure: HTTP protocol with added layer of security in form of SSL encryption. 23

HUD heads-up-display: display portraying information in the users view, while not making the user have to look away to see it. 27

MobX JavaScript library that tries to simplify state management, by applying functional reactive programming. 11

Proxy server application that acts as intermediary between clients and server, that want to communicate. 23, 35, 36

React JavaScript library developed by facebook; used to build user interfaces. 11

REST Representational State Transfer; programming paradigm for distributed systems; commonly used to describe webservices. 30

URL Uniform Resource Locator. 10, 27, 29–32, 35, 36, 39

vertex a vertex (plural vertices): point in 3D space where two or more lines meet. In 3D rendering vertices are used to describe 3D objects and their faces.. 4, 16

XML Extensible Markup Language: language to describe hierarchically structured data. 9

1 Introduction

1.1 Motivation

Virtual reality (VR) has gained more and more popularity over the past years. Its possibilities have developed a lot and its technology is improving further and further. Applications that utilize virtual reality can be found in any field. Doctors use it, to train for surgeries. Pilots use it to simulate their flights. Mechanics use it as a documentation and simulate the steps needed to repair big machines. VR can be used for gaming, learning, communicating and exploring, without having the need to leave the house. With its introduction into web technologies, a whole new dimension of possibilities has been added. Together with the ever improving technology of smartphones, virtual reality got increasingly accessible for the the broad mass. This enables VR developers to think bigger than before and create projects that are more focused day to day struggles and connecting people, than on specific applications. One big topic when creating such a VR space is the combination of 2D and 3D interfaces. The internet has become a big part of the human life and contains an immense amount of information and technologies. But those are only available in 2D. To bring this into virtual reality environments, you have to find a way to combine those two worlds into one. This thesis tries to develop and implement methods, that make the usage of regular online websites available in VR.

1.2 What is VR?

VR describes a computer generated world. This world can be experienced, with specific VR hardware, such as the *Oculus Rift* or the *HTC Vive*. Opposed to other extended reality (XR) experiences, like augmented reality (AR) or mixed reality (MR), virtual reality tries to completely detach you from the physical world. Its motto is immersion. The VR headsets, often including earphones, make you dive in into an alternate reality, without having much sense of what is happening outside of it. Due to its capabilities, of making the scenario feel very real, it is often used to simulate extreme situations. These simulations are then used to train people how to act, when confronted with such circumstances. Present-day technology reduces the need of having to buy dedicated

VR hardware, to experience virtual realities. Modern smartphones mounted to your head can also function as a VR device, making the experience more accessible.

1.3 Rendering 101

In this section basic rendering techniques are explained. They help understand what exactly is happening during the rendering process.

1.3.1 The world of spaces

In order to be able to see 3D objects on a 2D screens, a lot of calculations have to be made. Calculations mostly regarding coordinates. Those coordinates have a long road to go until they lead to a pixel being displayed on the screen. They start in object space, which is the space defined by the Designer of an object. Let's say a tea can is designed. Then the center of the tea can is the origin of its coordinate system. Relative to this system you then start building your vertices describing the tea can. Vertices are points/vectors in 3D space which when connected, create a wireframe of the desired object. Finally the finished tea can file, contains multiple vertices relative to said object space (coordinate system). Upon placing multiple of those tea cans into a world, you need to be able to compare the vertex positions from one can with vertices of another can, in order to calculate lighting and collision etc. Thus every vertex of the tea can has to be transformed into world space, by multiplying their positions with the a so called *worldMatrix*, as depicted in Figure 1.1 Afterwards all vertices of all the tea are relative to a new coordinate system (world space), which represents the whole world you are working in. In order to render the scene from a specific perspective, a camera object is placed somewhere in the world. To simplify the subsequent tasks like applying lighting or calculating occlusion, you transform all objects in world space into a new coordinate system. In this system, the camera is the origin and the negative z-axis represents the direction the camera is looking at. This specific transformation is shown in Figure 1.2. The new coordinate system, obtained after the transformation, is called view space, also known as camera space.

1.3.2 Camera Projection

The view frustum from a camera in 3D space is defined by the near and the far plane, shown by the light blue part displayed in Figure 1.2. It defines the visible space. Everything outside of it will not be displayed. In order to get the visible part to show on screen you first have to transform the frustum into a unit cube with coordinates from -1 to 1. This step is called *Camera Projection* or only *Projection*. There are multiple types of

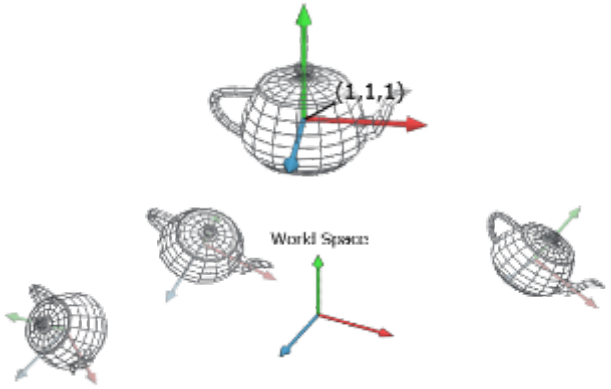

Figure 1.1: The Image shows a tea cup mesh in object space, being transformed multiple times into world space. Taken and altered from: [11]

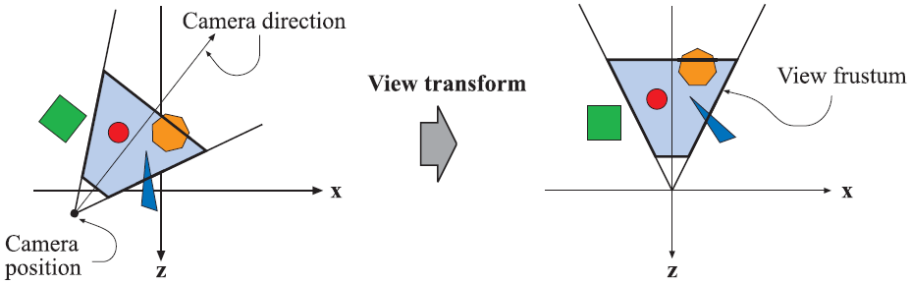

Figure 1.2: In the left illustration, the camera is located and oriented as the user wants it to be. The view transform relocates the camera at the origin looking along the negative z-axis, as shown on the right. This is done to make clipping and projection operations simpler and faster. The light gray area is the view volume. Here, perspective viewing is assumed, since the view volume is a frustum. Similar techniques apply to any kind of projection. Image and caption taken from [18, p. 17, Figure 2.4]

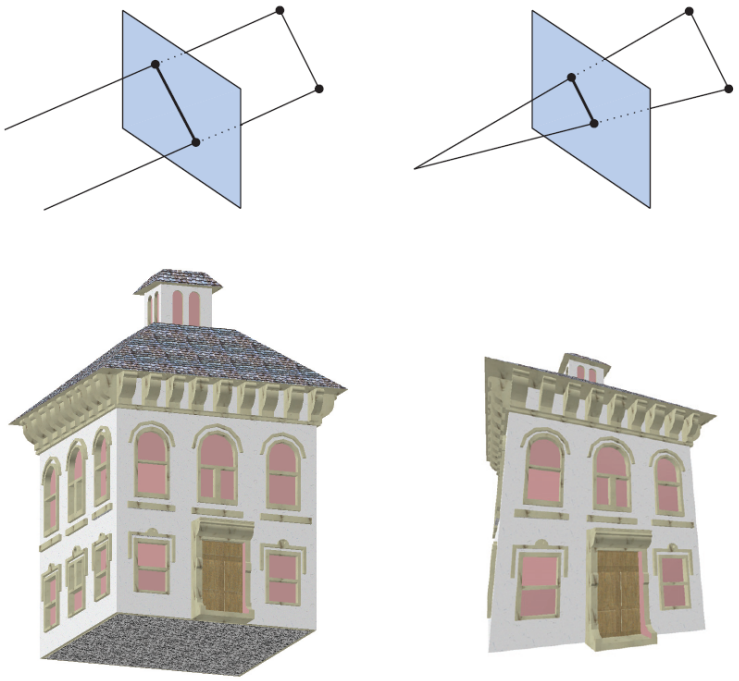

Figure 1.3: The left side depicts an orthographic, or parallel, projection; the right side shows a perspective projection. Image taken from: [18, p. 18, Figure 2.5]

cameras, with different calculation steps in order to receive the camera projection. The most common camera is the perspective camera, which mimics the human perception, by making objects farther away from the camera look smaller. That is the reason the frustum in Figure 1.2 is formed like a cone, opposed to the frustum of an orthographic camera, which is represented by a cuboid. The difference between those two camera projections can be seen in Figure 1.3. As already explained the projection consists of transforming the view frustum into a unit cube. This is done by matrix transformation as explained in subsection 1.3.3. After the projection the coordinates of the objects in the scene have normalized device coordinates. While for orthographic cameras, this process is trivial, perspective camera frustums have to be squished, inorder to fit the desired output. Even though it is a transformation from one volume into another, it is called projection, because the resulting image does not include z-coordinates. This makes it a projection from 3D to 2D space. [18, p. 19]

1.3.3 3D Transformation

3D transformation consists of three main procedures: translation, scaling and rotation. Translation moves an object from one position to another, while scaling makes it bigger or smaller and rotation rotates it around a certain axis. All three types of transformation are realized via matrix calculation. This technique is very common in games or other applications that are based on 3D rendering. In those types of applications, the above procedures are applied to objects in 3D space.

The way this works is by extending the vectors of the objects vertices by one extra dimension. So the earlier three dimensional vectors, become four dimensional, while the forth coordinate always is a 1. These extended vectors are then multiplied by the according *transformation matrix*, which is a 4x4 matrix. Depending on what desired outcome is, your matrix looks different. For example when scaling an object, the diagonal of the matrix is filled with the corresponding factors for the x, y and z value, as displayed in Figure 1.4. A *translation matrix* on the other hand, has the corresponding offset values of the three axes, placed on the bottom row. Opposed to the other two transformation methods, *rotation matrices* are axis specific and thus there are three of them.

Objects often need to be transformed in multiple ways. This can be accomplished, by combining the mentioned matrices in a matrix multiplication. To assure the correct behaviour of the transformation, you first apply the scaling, then the rotation and finally the translation. Which leads to this equation: $RM = TM * RM * SM$, where SM is the scaling matrix, RM is the rotation matrix and TM is the translation matrix. Applying the resulting matrix to the vertices of an object, leads to the same result as applying each of transformation matrices, by itself, in a sequence. This characteristic is used to create a *worldMatrix*, which transforms an object from object space into world space, as described in subsection 1.3.1.

1.3.4 Blending

Blending is one of the last steps in a rendering pipeline and is used to calculate the effect of transparent objects on the scene. It takes place after all lights, shadows, textures etc. have been applied to the objects and each part of the scene is mapped to the exact pixel it represents. So what you have is a big map of pixels containing an array of color values for the object colors at the pixel position. Those color values per pixel are also called *fragments*. The so called blending is the process of traversing through the fragments of each pixel and combining their color values according to the alpha

$$[x \quad y \quad z \quad 1] \mapsto [kx \quad ky \quad kz \quad 1]$$

$$S_k = \begin{bmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D scaling matrix

Figure 1.4: The image shows the scaling of a vector by a 3D scaling matrix. The matrix scales all three values, x, y and z by the same factor k. Image taken from [11]

values. The alpha values indicate the transparency of an object material. Staring at one pixel, the *fragment* that is the farthest away is chosen and the current pixel color is overwritten with the *fragments* color. Now each *fragment* of the pixel is trespassed in order of their z values and the current pixel color is combined with the *fragment* color. The resulting color is then set as new pixel color. This combination can happen in multiple ways and is completely programmable. The most common approaches are, to either completely ignore the previous value and always overwrite the pixel value with the closer fragment color, or to use the alpha values of the fragments to implement some transparency. One basic way to do this is to use the alpha value directly as the color factor. The corresponding equation looks like this:

$$C = \alpha * B + (1 - \alpha) * A$$

Where A is the previous pixel color, B is the current fragment color, α is the alpha value of the current fragment and C is the new pixel color value. When applying this equation for each fragment most colors still will get overwritten, because the fragments mostly have ones as alpha values. This means the second product equates to 0. But for example a red window would let the previous color shine through and still add some of its red to the final pixel color. Summarized: blending describes the process of the fragment combination and in most cases is completely programmable. [18, p. 23-24]

1.4 The internet and its technologies

This section introduces modern web technologies, their advantages and limitations.

1.4.1 HTML Events

Most of the times when creating a web application a reaction to user input is desired. In order to realize that, you first have to be aware of what the user does. This can be accomplished by listening to DOM events sent by the browser. Each user interaction and even other interesting things, like the change of the network status, triggers an events. Each element of the HTML document can be extended by event attributes. These attribute values can be filled with a callback method, which gets executed every time the corresponding event occurs. For example if you want to track where the mouse is at any given time. The desired event you want to listen to in that case, is the *mousemove* event. It gets triggered every time the mouse is moved by the user. By adding the *onmousemove* attribute to the documents body element and setting the value of that attribute to a callback method in your code, you can listen to the event. Alternatively this *eventListeners* can be added directly do the root document. Those listeners also are initialized with a callback method. Now every time a *mousemove* event is triggered the passed method will be executed. The executed code additionally receives the current mouse state in form of a *MouseEvent* object. This object contains information like the position of the mouse relative to the client (browser), screen and page. It also contains information like how far the mouse is away from its previous position, which mouse buttons are currently pressed and even if keys like *ctrl* or *alt* are pressed. [12] Your callback methods then can react accordingly to the given information. Other mouse related events like *mousedown* or *dblclick* all deliver *MouseEvents*, while other events like for example the *wheel* event, deliver a different event object called *WheelEvent*. Opposed to pixel values for the mouse position it contains the delta values of the mouse wheel, describing how much the wheel has been turned. All those events enable you to track most of the users interactions and react to input. Events also can be triggered manually via JavaScript by creating a corresponding event object in the code. Afterwards the DOM element that you want to react to the created event has to be obtained from the document. Lastly the element can be told, to dispatch the event, by passing the event object into a *dispatchEvent()* function. This will trigger all listeners listening to given event on that element. Note that events only only can be triggered directly on an element. They cannot be dispatched directly on the browser.

1.4.2 Cascading Style Sheets

Cascading Style Sheets also known as CSS is a styling language used to describe the rendering of structured documents, such as HTML or XML, on screen and paper. [3] It has been developed in multiple small modules and can describe the layout an appearance of a document in many ways. For example you can set the fonts and colors

of texts or define the position of elements. Even placement of objects in 3D space is possible, by using the CSS 3D transformation module. One of the more recent modules, called flexible box module solves a lot of scaling issues, by removing the need of setting exact pixel values for positions. Instead the parent properties describing the desired behaviour can be set and the browser takes care of the positioning. The advantage of such a styling language is that is detached from the actual content. This way both, the styling and the content become easier to maintain and adjust. Furthermore the cascading part is very helpful, it makes the children of a document element inherit the parents styles. This removes the need of duplicating styles for each child and in general reduces the amount of styling code needed.

1.4.3 Same origin policy

"*The same-origin policy is a critical security mechanism that restricts how a document or script loaded from one origin can interact with a resource from another origin. It helps isolate potentially malicious documents, reducing possible attack vectors.*"[15] An origin is defined by the host, the protocol and the port (if specified). So two URL's have the same origin, if all three properties are the same. If one of them differs, the interaction between those two resources is called cross origin. Cross origin interactions underlie specific regulations determined by the same-origin policy. For example reading from a cross origin is mostly disallowed, while embedding and writing often still is possible. Embedding also can be influenced by the site, by using HTTP header called *X-Frame-Options* and *Access-Control-Allow-Origin*. The *X-Frame-Options* header defines wether a site wants to be embedded in general or not. In the *Access-Control-Allow-Origin* a site can define origins, which are allowed to share cross origin resources. But even when embedding is allowed, the same-origin policy strongly limits the API access to the embedded HTML document, since you most probably are not on their whitelist to CORS access. The actual limitations depend on the browser, but for example Firefox only allows four functions on a cross origin window element. *Blur*, *focus*, *close* and *postMessage*. Furthermore attributes, except for the location are read only. When trying to use other API methods or modify the attributes it leads to an error, which also is protected. It can't be interpreted in JavaScript. The code leading to the error just receives a generic error. The actual error message is only visible for the user in the console. Additionally the browser storage is separated by origin and JavaScript code from one origin cannot access the storage from another origin. [15] All of this is to prevent phishing and other malicious attacks from happening and most other browsers have similar implementations of the same origin policy.

1.4.4 Vue.js

"*Vue (pronounced /vju:/, like view) is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries.*" [9]

It is a component based JavaScript framework, like React JS or Angular JS. Your own, so called Vue components, can be built, which can receive properties and implement component specific logic. The components consist of the HTML part, where you can insert any type of HTML element, like a <*button*>, <*image*> or even other Vue components. Those components can then be inserted into the body of the HTML document. A big advantage of Vue are the built in data bindings. For example an object containing a text value can be passed as a property. It then can be bound to a <*label*> element in the component, so the text is displayed. When changing the text of the object through any other code, the component notices the change and immediately updates the value displayed in the component. This shortens the code base, since property updates do not have to handled by yourself. Other frameworks like React need external libraries like MobX to accomplish the same thing. Vue.js is very light weight, but still powerful, so a good basis to build an web application on.

1.4.5 Flux architecture

"*Flux is the application architecture that Facebook uses for building client-side web applications.*" [7] It assumes a global application state that is accessible from all components responsible for the rendering, which most single page application have. Flux is based on four main components: Actions, a Dispatcher, one or more Stores and the View components. It enforces single direction data flow. The general structure can be observed in Figure 1.5.

Actions are small objects containing information about the user interaction. The Dispatcher is the distribution layer, that delivers all the actions to the stores. There's always only one dispatcher. Stores are responsible for the application state or parts of it and the only entity allowed to alter the state. Small applications only need one store, while big applications mostly use more stores that are logically separated from each other. View components are the components visible to the user. Since the architecture is from facebook, in most cases the components consist of facebooks React components,

but the architecture is applicable to any other UI component. The View components derive their local state from the application state and emit new actions when the user interacts with them. [7]

For example if a user changes text in an input field and presses enter. The View component containing the text field, reacts to the enter press. It calls an *action creator* for a *TextChangeAction* and passes the text id and the new text as a parameter. As the name implies, the *action creator* creates new instances of Actions, but it also can have more logic to it. Often *action creators* are in charge of interacting with the backend of the application. In such a cases, they first send a request to the server and only after receiving a positive response, they create the corresponding Action and dispatch it. In this example it would be a *TextChangeAction* containing the id and the new text. This Action is dispatched by passing it to the dispatch method of the Dispatcher. The Action is then distributed to all the subscribers subscribed to the Dispatcher. The subscribers consist of Stores. A Store can subscribe to a Dispatcher by calling the subscribe function and passing a callback as parameter. This callback is part of the Store and contains the logic to process the Actions received. Upon arrival of the *TextChangeAction* in the Store, the application state gets updated according to the Action. In this case, this would mean that the text for given id gets replaces by the new text. When finished a *changed* event is emmited by the Store. The View components listen to that *changed* event and retrieve the new data from the Store. Usually only a few big View components called *controller-views* are listening to the stores and pass the retrieved data to their children.[7]

The main purpose of this architecture is to simplify the data flow and lead to more transparency regarding the behaviour of the application. It removes two way bindings and thus also cascading updates, where one update leads to multiple other updates and so on. The application state is only maintained in the Stores, which makes it possible for the other components to be mostly decoupled from each other. That makes the system as a whole more predictable and modular. [7]

1.4.6 WebSocket

The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers. The protocol consists of an opening handshake followed by basic message framing, layered over TCP. The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers [...]"[6, p. 1] It's advantage over the common HTTP, is that there is no request response data flow. Instead data can

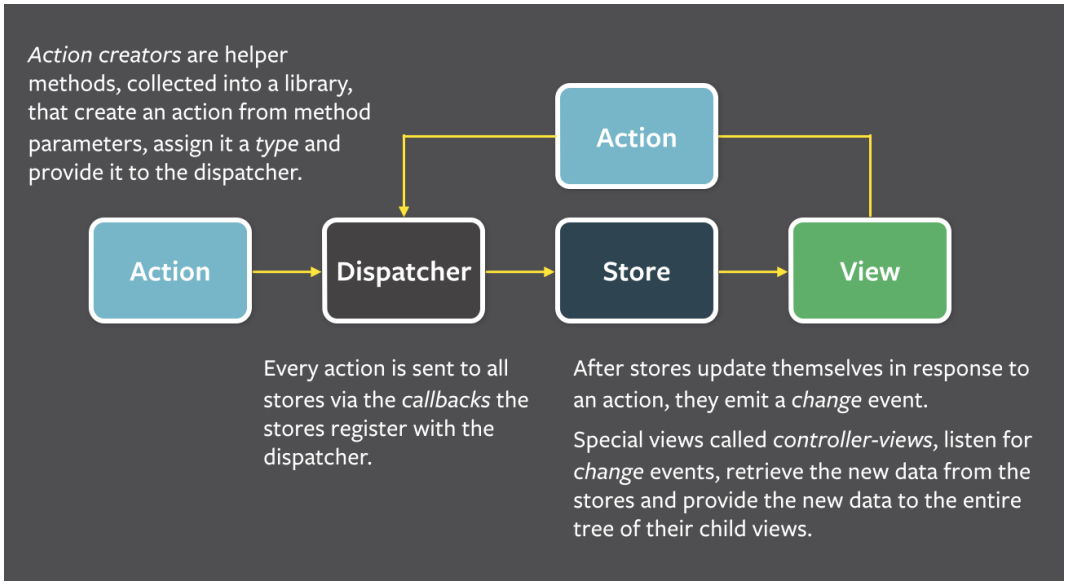

Figure 1.5: The image shows the abstract data flow of a application built with flux architecture. The arrows show the direction the data is flowing and each node has a text describing the general approach on how the data is processed in each step. Taken and resized from: [7]

pushed form party to party, whenever it is necessary. This makes it easier to keep client and server states consistent. For example when having a server state that can be influenced by multiple clients at the same time. Those clients would have to keep polling with HTTP requests, to know if something has been updated. With WebSocket the clients can be updated by the server whenever its state changes, without having to ask for it.

1.4.7 The Protocol Buffer Library

"*Protocol Buffers [(protobuf) are a] language-neutral, platform-neutral, extensible mechanism for serializing structured data*".[1] This mechanism has been developed by Google and claims to be a smaller, faster and simpler option compared to the Extensible Markup Language (XML) *"The protobuf library defines a fast binary format for messages. Message structures (so called 'descriptors') are defined in a simple C-like language in .proto files. An example is given in [Figure 1.6]. From these, interfaces for different programming languages can be generated using the provided extensible compiler, protoc. A given interface contains the code describing a set of classes providing the (de)serialisation functionality tailored for a given .proto file and platform. The descriptors themselves can be serialised as protobuf messages, facilitating the inspection of arbitrary serialised messages without needing the descriptors at compile time. In summary, the relevant features of the protobuf library are the following:*

- *Fast serialisation and de-serialisation of structured data in the form of messages*
- *Separation of data structure definition and code*
- *Extensible code generators for different programming languages*2
- *Messages can be nested*
- *Repeated fields can store data in a manner similar to dynamic arrays*
- *Content can be omitted from optional fields, and does not take up space in this case*
- *Thread safety*"[10]

1.4.8 WebGL

"*WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES, exposed to ECMAScript via the HTML5 Canvas element.*" [21]

In other words it is a OpenGL port for the browser, to make shader language accessible via JavaScript. With WebGL you can access the graphics card via JavaScript and make use of its calculation power, to display 3D scenarios in real time. The scenarios are visualized, by drawing on a HTML canvas element. This element has been introduce, to make it possible to draw via JavaScript.

```
1  syntax = "proto3";
2  package ubii.dataStructure;
3
4  message Vector3 {
5      double x = 1;
6      double y = 2;
7      double z = 3;
8  }
```

Figure 1.6: Example for a protobuf message definition for a three dimensional vector. Image retrieved vai screenshot from the ubii-msg-formats project. [2]

1.5 Serialization methods

1.5.1 JSON

JavasScript Object Notation, also known as *"JSON[,] is a lightweight, text-based, language-independent syntax for defining data interchange formats. It was derived from the ECMAScript programming language, but is programming language independent. JSON defines a small set of structuring rules for the portable representation of structured data."*[16] *"[It] is a syntax of braces, brackets, colons,[...] commas [...] [and] provides a simple notation for expressing collections of name/value pairs."* [17] JSON is used in web applications, to exchange data. A JSON construct can be attached to a HTTP request body and sent together with the request. As the name implies, the JavaScript Object Notation (JSON) is based on JavaScript objects. This makes for a native support by the programming language. Every JavaScript object can be serialized into JSON format and every string in JSON format can be deserialized into a JavaScript object, making JSON a perfect tool for web applications, that do not have fixed typing.

2 Related Work

2.1 three.js

"*Three.js is a 3D library that tries to make it as easy as possible to get 3D content on a webpage.*" [19] It is based on WebGL and tries to minimize the necessary work to set up and work with 3D scenes. Thus removing the need for low level API calls directly on the WebGL interface. "*It handles stuff like scenes, lights, shadows, materials, textures, 3d math, all things that you'd have to write yourself if you were to use WebGL directly.*" [19] Everything combined creates a interactive 3D world displayable in the browser. To understand how three.js works its structure has to be observed.

An example for a typical three.js setup can be seen in Figure 2.1. The base of such a setup are geometries and materials. Geometries store the vertex position for all vertices defining a shape for e.g a cube, a sphere, a plane, or more complex thing like a tea can. Materials consist of multiple parameters describing the surface of an object. They are defined by a texture or a simple color value and contain things like normal maps, opacity values, blending and lighting options. When combining a material with a geometry a mesh is obtained. Those meshes can either be directly added to a scene or combined to an object. Those objects can then be grouped even further. For example if you want a squadron of space ships to fly around a space ship geometry and a material describing its surface is needed. Then multiple meshes can be instantiated with those and be grouped into a squadron. This way all of the space ships can be moved at once just by moving the group. To see reflections when they fly by on the screen a light source has to be added. This source emits light and changes the way materials look, for example materials that try to mimic metallic surfaces, will reflect this light, leading to changes in the pixel colors. There are multiple types of lights like point lights, ambient lights and directional lights. These are used to replicate the real world lighting in a digital 3D space. All those objects combined define a scene.

After the scene has been generated, a camera of the desired type (mostly perspective) can be instantiated. This camera together with the scene is then passed to a renderer. A renderer is instantiated with some render options defining if antialising is enabled, or alpha values shall be considered in the color calculations. Also the width

and height of renderer has to be set to the desired output picture size in pixels. Upon render all objects in the scenes and their meshes are put through a complete rendering piepeline. Meaning that they get transformed into camera space, projected into a unit cube and rasterized. Additionally lighting etc. is applied. Finally the color values for each pixel are calculated and drawn on a HTML canvas element. There are multiple types of renderers but the most common one is the WebGLRenderer, which uses the WebGL API, described in subsection 1.4.8, to draw. Another renderer that comes with three.js is the CSS3DRenderer. This one is used, to display HTML elements in 3D space. Similar to the WebGLRenderer, it takes a scene and a camera into a render function and renders the scene accordingly. The main difference being, its rendering procedure. The scene it takes doesn't consist of Meshes or lights, it solely consists of CSS3DObjects. CSS3DObjects are inherited from the base Object3D but instead of being placed using the usual matrix transformation, the objects rea transformed via CSS 3D transformation. This is the reason the scale of CSS3D scenes can differ from WebGL scenes. CSS uses pixel values as measure, while WebGL has its internal measurement system which tries to mimic the metric system. More about CSS can be read in subsection 1.4.2. The CSS3DRenderer also is not responsible for rendering anything itself. Its only purpose is to position and scale the HTML elements in the scene relative to the camera. The result is then attached to the DOM. The actual visualization is performed by the browser. Another thing to mention is the fact, that every object in a scene is inherited from the Object3D class. So meshes, lights and even cameras are Object3D instances, with some more functionality on top. This way every object can be attached to every other object. For example when having a moving object in the scene, a camera can be attached to it, to make the displayed image update according to to movement of the object. The other way around is also possible. Objects can be attached to cameras, which makes them move around with them. This is specifically useful in VR, for making something always visible in the users peripheral.

2.2 A-FRAME

"*A-Frame is a web framework for building virtual reality (VR) experiences. A-Frame is based on top of HTML, making it simple to get started. But A-Frame is not just a 3D scene graph or a markup language; the core is a powerful entity-component framework that provides a declarative, extensible, and composable structure to three.js. Originally conceived within Mozilla and now maintained by the co-creators of A-Frame within Supermedium, A-Frame was developed to be an easy yet powerful way to develop VR content. As an independent open source project, A-Frame has grown to be one of the largest VR communities. A-Frame supports most VR headsets such as Vive, Rift, Windows Mixed Reality, Daydream, GearVR, Cardboard, Oculus Go, and can even*

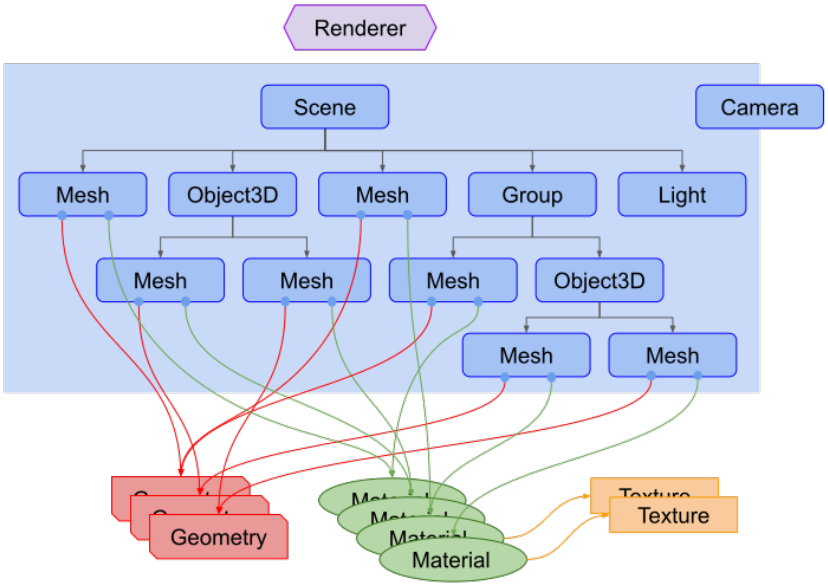

Figure 2.1: This Image shows the an exemplary structure of a three.js project, displayed as a tree. It consists of a renderer, a scene, multiple objects, meshes and a camera. Each mesh consists of a material geometry pair, while the material can be defined by a texture. Image taken from [19]

be used for augmented reality. Although A-Frame supports the whole spectrum, A-Frame aims to define fully immersive interactive VR experiences that go beyond basic 360[-degree] content, making full use of positional tracking and controllers." [8] It is one of the best frameworks currently on the market, to develop VR content for the web. Built on top of three.js A-Frame tries to simplify the implementation of VR worlds. Features like a built-in visual inspector, where you can move and observe the objects in an A-Frame scene, make the development process a lot easier.

2.3 mozilla hubs

Described in one sentence: "*Hubs is a virtual collaboration platform that runs in your browser. With Hubs you can create your own 3D spaces with a single click. Invite others to join using a URL. No installation or app store required.*" [23] It is an Open Source project initiated by the company Mozilla. Mozilla Hubs allows you to create you own Virtual Reality room and invite other people into it. Such a room has multiple features. The world is interactive, so objects can be moved, and even freehand paintings can be created with the pen tool. The main focus is communication and sharing. That's why there is a text chat, an emoji feature, a voice chat, and an option to share your desktop screen, or your webcam input. Additionally multiple video and streaming sources can be embedded, like YouTube, or Twitch. Those are aslo interactive, meaning a video can for example be started and stopped, or the volume can be modified. All those things are shareable with multiple people. Every room has its own link which can be sent to the persons you want to invite. Every member joining a hub can choose from a collection of predefined avatars, that will be visible for all other members and indicate the persons position. All members of the room can be managed by the room creator. Mozilla hubs has authentication and authorization implemented, so one person has the ownership of a room. Interaction restriction, like what object is allowed to be moved, can be defined per user from that person having ownership. [5] Mozilla also offers their own servers. They can be used to test the environment and meet with a small amount of people. If a bigger events with 25 or more people is planned, a room can be rented from Mozilla. But since it is Open Source, you can also simply host it yourself, by setting up the whole project on your own hardware.

The frontend or client side of this project is built with A-Frame and the underlying three.js. In order to enable user to user communication, a connection to two backend services is established. One is responsible for voice and video communication. It collects all video and voice output from the clients and distributes it to all other clients connected to a room. The other service takes care of the remaining interactions,

like drawing, moving objects etc. Both services consist of a big stack of different server technologies.

2.4 ubi-interact

Ubi- interact, also known as ubii, is a project developed by the "Forschungsgruppe Augmented Reality" (FAR) from the Technical University of Munich (TUM). Used as a framework for building reactive and distributed applications, it can send and receive data from any client connected to it. One advantage is its subscription feature, where topics can be created and subscribed to by clients. When a message regarding that topic is published by a client, all subscribers receive a WebSocket message, with the corresponding update. Furthermore ubi-interact implements Googles protocol buffers, making it easy to define additional data formats and accelerating the serialization and communication between clients. The framework allows for custom definitions of your systems behaviour, based on the received topic data. These so called interactions can be stored and retrieved from the backend and thus can be reused. Ubii is developed in a very modular way, making it possible for your implementations, to be decoupled from the specific devices or environments. By having context neutral interfaces, interactions become reusable in multiple environments and devices become exchangeable, which makes the framework as a whole very flexible. [20] Storage functionality is also brought, which cannot just store files locally, but also retrieve it from an online database. This storage is also easily extended, by custom implementations. But it does not just come with a backend. There is a fully functioning frontend developed by FAR too. It contains multiple former projects in form of applications. For example a VR Keyboard implementation, where the smartphone can be used as a keyboard, while being in a virtual reality experience. Another example is the XRHub which implements a three.js room, and supports VR. This is the par, that will be further developed in the course of this thesis. The frontend also brings a lot of tools with it, like a topic inspector, or performance tests and even has isolated interface access. Interfaces like cameras, smartphose or the self developed *Ubii Controller* can be accessed and tested without any other code being involved. The frontend is built with Vue.js components and a lot of the applications use three.js for the visualization. The last part of the project is the *ubii-msg-formats* git repository. There the data formats and interactions are defined in .proto files, which later on can be used by the back- and frontend, to serialize and distinguish the individual messages. This is also where new data types can be introduced.

3 The XR-Hub

3.1 Selecting the right framework

Planning a Software project is similar to planning a house. While of course it is important to plan for doors, windows and lights, the most important part is the foundation and the materials you're using. A house is built to last for ages it should be built with as little hurdles as possible. Thus the ground on which your house is placed and the type of bricks your house will be made of, has to be chosen wisely. Same goes for the Software. Since the code should be designed to last as long as possible, without having to adjust it, a solid foundation is needed, which gives the needed capabilities of building a good software on it.

One very promising candidate as a ground to build on, was mozilla hubs. It already implements most of the features we need for a collaboration tool and even more. Features like drawing, voice chatting, video, desktop and camera sharing are all features that fit exactly what we are triying to build. So why not use it?. Of course the idea behind hubs is very similar to ours. The fact that it is Open Source and the community is actively developing it, also is a reason to go for it. Most of the features they have implemented, also make sense for this exact use case. But one main feature we want to implement is missing. The interaction with websites. In hubs only videos and streams are able to be embedded. Other sites, like Google Maps for example, are not supported. But that should not be a drawback, since we wanted to implement that feature anyways. The main reason, that lead to not using mozilla hubs, was its project size. The client side code let alone consists of over 221 thousand lines of code (number from 11.08.2020, retrieved by using the GitHub Gloc Chrome extension on the official GitHub repository), not mentioning the two backends. And to work on it locally, all three applications have to be running on your machine. In order to even analyze if the features we want to implement are realizable in mozilla hubs, it would take weeks of going through the code and understanding it. With the possibility that we come to the conclusion, that it does not work. Thus the decision to leave mozilla hubs for now and create a smaller scale prototype has been made. When having success with the prototype, mozilla hubs could still be extended with it.

So instead we chose ubi-interact as our foundation. As mentioned in section 2.4, it is designed for applications with external devices like vr-headsets and works with modern technologies like the protbuf protocol from google to spread the interactions from the user to other clients. So it does not just have a working frontend, which is easily extendable, it also comes with a fully functioning backend, that also can be adjusted to our need. The main part used, was the already existing XRHub application inside of the ubi-interact frontend, which consists of a small three.js scene with VR capabilities. Furthermore the already implemented routing and the backend has been used. The main features utilized from the backend, were the topic subscription, the storage and the services. So a solid ground has been set up. Now the right tools for building have to be chosen. The goal so to build a VR-Application in the browser, so we are limited to browser technology, which luckily is not such a big limitation nowadays. There are a few established frameworks that support VR-developement. Libraries like A-Frame ReactVR, or WebXR are all valid candidates, that have a lot to offer. Since the ubi-interact frontend was built with Vue.js components, ReactVR would mean a break in the architecture, so it was not the favoured solution. A-Frame is very focused on the VR part and has little possibilities, to display other websites or HTML elements within a scene. It still comes with a lot of advantages like its scene inspector and its big community, so it also is a valid way to go, but at the time being, we didn't see it as superior to three.js. This was due to the fact, that we would have to use three.js anyways, to be able to display the HTML part. This leaves us with WebXR and three.js. WebXR is a pretty low level API on WebGL, so it would mean a lot of work, a lot of other people already had done, to setup a working VR Scene. This lead to the decision to use the three.js library, which is built on top of the WebXR standards, instead of reinventing the wheel. Further Vue.js components were used to realize, buttons, links or text boxes inside the scene.

3.2 WebXR, WebVR and browsers

"*Hardware that enables Virtual Reality (VR) and Augmented Reality (AR) applications are now broadly available to consumers, offering an immersive computing platform with both new opportunities and challenges. The ability to interact directly with immersive hardware is critical to ensuring that the web is well equipped to operate as a first-class citizen in this environment. Immersive computing introduces strict requirements for high-precision, low-latency communication in order to deliver an acceptable experience. It also brings unique security concerns for a platform like the web. The WebXR Device API provides the interfaces necessary to enable developers to build compelling, comfortable, and safe immersive applications on the web across a wide variety of hardware form factors.*"[13] It is the successor of the WebVR API, which

"*[...] provides purpose-built interfaces to VR hardware to allow developers to build compelling, comfortable VR experiences.*" [22] The WebVR development has halted in favor of being replaced by the WebXR Device API, but most browsers still support WebVR while WebXR is in development. [22]

Right now we are in between two API's. One is deprecated and will be replaced soon and the other is under developement and subject to change at any time. This makes it difficult to develop VR content. Most frameworks like three.js already hopped onto the WebXR train, only implementing WebVR with older versions. Three.js for example remove WebVR support colpletely, since december 2019. [14] Most browsers on the other hand, do not support WebXR yet. Firefox and Chrome have some flags and settings WebXR features can be enabled, but those are experimental and very unstable. Plus even the way those features are enabled has changed from time to time. Firefox tries to keep it up to date and also developed a WebXR Device emulator for people that do not have access to XR-Devices, but it still does not work every time. Firefox also has the downside of having updated their same-origin policies recently, which makes it impossible to connect to the ubii backend without refactoring a big portion of the existing code and using a reverse Proxy.

Another hurdle is the security policy regarding WebXR. Since the ability to track the whole body movement can be abused and used in a malicious way. Browsers lock the XR features for normal http connections and only enables them if a secure connection is established. Which means the client has to be delivered via HTTPS. This has been solved by using a self signed certificate and delivering the built project via nginx, which is a web server software. Nginx supports HTTPS, making it possible to use the WebXR features. This solution also influenced the development, since to test your code the whole application has to be built. It only takes around ten seconds, but compared to the sub one second recompile time in the development mode, it is big difference.

During the process of writing this Thesis, most of the time it wasn't possible to develop for VR. Either because of the COVID-19 pandemic limiting the access to the necessary devices, or because of the fragile browser support. This is the reason, that the usage of VR in this project is theoretically possible, but no VR interaction has been tested, nor implemented. Which means the scene can be observed through a VR headset if a working browser setup is found, but no interaction with it is possible. The WebXR support situation is subject to change and when the API stabilized it will be very easy to develop for VR, but as of now it is not practicable.

3.3 Combining websites and VR

The main idea of the project is, the ability to insert websites from multiple sources into a VR Scene and interact with them. This scene should then be accessible for all invited persons, to also have a look at the displayed websites. So first of all way to display a website inside a three.js scene has to be developed. Three.js already comes with a functionality to display CSS content in a scene with its CSS3DRenderer and CSS3DObjects, but those on their own do not support VR. In order have the ability to ender VR and move around on virtual ground a separate scene using a WebGLRenderer has to be created. The issue is how to combine those two. Since two different scenes are involved, the objects in one scene do not interact with the objects in the other scene. Either the CSS3DObjects are not visible at all, or they cover all other Objects3D objects in front and behind them, no matter where they are placed.

To make those CSS3DObjects look like they belong to the WebGL-scene, you can use a trick as described in this article [4]. The trick and how it works is explained in the following part. First of all a plane has be created in the WebGL-scene, that plane is made transparent by setting the opacity of its material to zero. Now a CSS3DObject is placed at the exact same place in the CSS3DScene. Since the plane is transparent, the CSS3DObject is visible through the plane. The problem is, when another object in the WebGLScene is located behind that plane, it also appears through it, because of the opacity of the plane and the fact that the WebGL renderer has priority. That is not what the desired outcome. The plane should represent the website and only the website, so it looks like the website is a texture on the plane in the WebGL scene. To accomplish that, a three.js blending option is used. As explained in subsection 1.3.4, blending is a calculation, to determine pixel colors. Three.js has the "THREE.NoBlending"-option for materials, which changes the blending calculations for that material. When now calculating the pixel color, by iterating through the fragment colors, fragments with the "NoBlending"-material option set are treated differently. Those fragment colors are not combined with the previous calculated color according to its alpha value, instead they overwrite the previous color. This leads to the desired behaviour in our WebGL scene. The only thing left to do, is setting the material color of the WebGL plane to black. This is to prevent alteration of the colors of the website or other transparent objects in the scene, placed in front of the plane. So the rendering problem is solved, but there are more issues that have to be tackled. For example the website should be able to move and not be stuck in one place. The ability to move and rotate the websites is needed. In order to gain this ability, both the CSS3DObject and the WebGL-plane, need to be able to move simultaneously. This is realized by the ThreeWebsiteCanvas class, which creates the previously discussed plane together with a CSS3D object and

makes them act as one. Another point to mention, is the scaling of the CSS3D object. The scales in a CSS3D scene differ from the ones in a WebGLScene. That has to be taken into account. So when scaling the plane to x and y the CSS3D object cannot be scaled with the same values. Width and height of the iframe have to be considered too. This has been solved by introducing a resolution variable, which stores the iframes dimensions. When the CSS3D object containing the iframe is scaled, the x value is multiplied with `1/resolution.x` and the y value with `1/resolution.y`, which makes it look the same size as its WebGL counterpart. After all these adjustments a website could be appreciated via a VRHeadset, as long as VR is available.

3.4 Interacting with the scene

Even though the base for moving and rotating the websites has been set, it is still not possible, since there is no way to interact with the scene. This still has to be implemented. The application should be usable by multiple devices not just VRHeadsets, so the most common interaction has to be implemented too. The mouse and keyboard interaction. First of all the mouse movement has to be tracked. Fortunately the standard browser MouseEvents, described in subsection 1.4.1, can be used to do so. Together with *eventListener*s custom callbacks can be hooked onto those events. Those listeners execute the method defined and passes the event that triggered them, to it. This way the mouse state, at the time the event was triggered, can be tracked by the application. This information now has to be used.

First of all comes navigation. There is no ability to navigate in the scene yet, which means a user is stuck with standing at the spawn location. For the mouse keyboard interaction, the FirstPersonControls.js, which already existed in the ubi-interact project, has been used with minor adjustments to the key mapping. This file is an example file from three.js showing how controls can be implemented. As the name already implies its behaviour is leaned on the navigation of first person games. So the "WASD"-keys are used to move forward, left, backward and right. The right mouse button combined with mouse movement is used to turn the camera, which also affects the direction you move with "WASD".

Further, the decision has been made to use an actual Obejct3D in the WebGLScene to represent the mouse position. This way most part of the logic can be kept as similar to the VR interaction, as possible. In VR instead of having a mouse, two controllers function as input method, which also are represented in the scene via objects. To map the mouse position of the event to scene coordinates a few calculations have to be made.

Three.js fortunately already brings a lot to the table, for example the camera has an `unproject()` function, which transforms normalized device coordinates mentioned in subsection 1.3.2 into a position in world space. In order to be able to use this function, the MouseEvent coordinates first have to be converted into normalized device coordinates (NDC). The MouseEvent coordinates are given in pixels, from the upper left corner of the page, by substracting the offset to the parent HTML element from them, normalizing them and shifting the range from 0 to 1 to -1 to 1 the needed NDC's are obtained. To fix the distance the mouse sphere has to the camera, those coordinates are unproject and then subtracted by the camera position. The resulting vector is then normalized and multiplied by a fixed scalar. This calculated position is then applied to the mouse sphere. When those calculations are applied in the *eventListener* callback for the *mousemove* events, the mouse sphere becomes a real time representation of the mouse position inside the WebGL scene.

From that point on the objects the user is pointing at can be determined, by using raycasting. It works by sending a ray from the camera position through the mouse sphere position into the scene and collecting all objects intersecting with the ray. The first one in that list will be the mouse sphere, but any other object after that is the one the user is pointing at. Sending the ray also can be done without the mouse sphere just by sending the ray into the direction calculated before, but this isn't the only purpose of the sphere. It is used to enable the user to move objects. There are multiple ways to realize movement in a 3D scene. A move mode can be implemented where the three axes of the scene are displayed via arrows from the center of an object. Pull those arrows leads to a movement along the corresponding axis. Combined with a display for the actual position numbers, this is perfect for exact placement, but not really intuitive and quick to use, when just a rough positioning is desired. The other way is to make objects abled to be grabbed. Those grabbed objects can then be moved by moving the mouse. Since we do not expect the user to need a high amount of precision placing the websites the second approach was preferred. But this could idea still could be realized, by only using the mouse offset given from the mouse move events, without using an actual object in the scene. So why the mouse sphere? The idea was to stay as close as possible to the way it would be implemented in VR, in order to be able to reuse code and already experiment with the way it works. In VR the plan was to attach the object that the user wants to move, to the controller pointing to it. That way the controller will act as an extended arm enabling the user to precisely place the objects where he wants. So to replicate that behaviour the mouse sphere got implemented. When a user grabs an object its position gets transformed to the object space of the mouse sphere. Then it is attached as a child to the mouse sphere. So when the mouse is moved around, the user does not just update the mouse sphere position, but also the object position,

because it is a child object of the sphere. After ending the grab, the transformation from before is undone, so the object is back in world space and then the object is detached from the mouse sphere. As a side note: every other Object3D object would've done the trick as mouse representation in the scene. A sphere has been used because it also can be used as a visualization.

With that implementation, objects can be moved around, but one issue stays. All those calculations etc. are only done in the WebGL scene. What about the CSS3DObjects? Those do not move at all. This is solved by replicating the same logic in the CSS3DScene, i.e. another mouse sphere is created for the CSS3D scene and position is kept in sync with the sphere of the WebGL scene. But there is one difference, when reacting to grab actions from the user. The fact, that every object in the WebGL scene maps to exactly one object in the CSS3D scene, can be used as advantage. Instead of having to use the raycast technique in the CSS3D scene and to determine which CSS3D object is getting grabbed, a reference of the corresponding CSS3D object is attached to every WebGL plane representing a website. This has multiple advantages. First of all it reduces the amount of calculations made, which is always good when dealing with *mousemove* events. Those events fire very often, so you want to keep the calculations bound to such events at a minimum. Secondly possible future bugs can be eliminated, by making sure the objects picked are the ones belonging together.

More advantages can be observed when looking at the method used to change the URL of a website. That is an important feature to enable the user to change the content of a website canvas. There are multiple solutions on how to implement such a feature. For example a text box could be placed above the website containing the URL, or an overlay like in a HUD could be created, which always displays the URL of the website the user is pointing at. Since that action has to do with text it is very important that the user is able to read it. This can be an issue in VR. You do not always have the optimal distance to a website and if its canvas is tilted sideways to the user it becomes even worse. So the text box option is not a good solution. The HUD idea would solve the problem of the readability, but it can be difficult for the user to associate the information displayed in the HUD with the object in the scene. So both ideas are not optimal. The solution lastly implemented is something in between. Instead of a HUD, there is a ThreeConfigCanvas, which is displayed similar to the websites. It also consists of a WebGL plane and a CSS3D object, but opposed to the website canvas it does not contain an iframe, it is filled with a self made Vue.js component. This component contains a text box filled with the URL and a reload button. The ThreeConfigCanvas only exists once per room and is only visible when toggled. It can be toggled with a WebGL website object as a parameter, which is used to calculate the desired position. The ThreeConfigCanvas

lays itself on top of the website, tilts itself towards the camera and always has a fixed distance to the camera, to ensure readability.

Another design decision was made, for the movement of the websites. Moving objects is already implemented, but what is quite difficult, is to rotate them into the desired position. Due to this fact handles got introduced. One handle for general movement and one for rotation. The movement one is a simple block placed above the website and the rotation one is located below the website represented by a thin cylinder. So instead of grabbing the website directly, you grab one of those handles and either turn the plane or move it around. Those handles do not just simplify the rotation of the website they also visually separate the interaction with the 3D object of the website from the interaction with the actual iframe.

This leads to the most important feature. The interaction with the website itself. When implementing the website canvases as described in section 3.3, they are displayed but not interactive. The iframe content cannot be altered in any form. This is due to the way the browser events work. Since the parent of the iframe, namely the three.js scene already interprets all pointer events, the iframe itself does not notice any user input. The initial idea was to retrigger every event manually directly onto the iframe. It works by taking advantage of the raycast, that is used to determine where the mouse is pointing at. It does not just store the objects it intersects with, but also the position, where in the scene the intersection took place. This information can be used to calculate an exact position on the iframe. That point defines the element on the iframe from which the event has to be manually triggered. Now the event object just has to be duplicated and the desired element has to be retreived from the iframe. The event then lastly has to be dispatched on the retrieved element. The standard approach to dispatch a mouse event via JavaScript, is by getting the element you want to react to the mouse event from the DOM and dispatching the event via the element. Since exact position is know, the document function `elementFromPoint(x, y)` can be used, to get the desired element. With that element the event can then be dispatched as described in subsection 1.4.1. But here is the clou. When working with websites with the same origin, everything works fine. But the moment you try to embed a website from a different source, e.g. youtube.com or google.com, you have to deal with the browsers security policies. Those are very strict regarding Same-origin-policy, as explained in subsection 1.4.3, and make it impossible to interact with elements inside a foreign iframe via JavaScript.

The next idea was, instead of emitting the event on top of an element, a click on a certain position could be emulated. This comes with two problems. First of all it

is not possible to simulate MouseEvents via JavaScript, in any way, without using an actual input device. Secondly the parent would still swallows the event, leaving the iframe with nothing to react to. This lead us to our final solution. The CSS style "pointer-events". When set to the value "none", this style disables all pointer events for that HTML element. So setting the the pointer-events style of the three.js scene to "none" prevents the mousemove and mousedown events from being swallowed. That way the iframe can receive the mouse events and react to them, making it possible to interact with it as expected. The downside of this approach is, that it breaks the whole navigation and movement controls. So the interaction with the iframe is fixed, but the ability to move objects and navigate in the scene has been lost. That's why the "Toggle Website Interaction"-button got introduced, which switches between a website interaction mode and a scene interaction mode, by either setting or removing the mentioned CSS-style. Even though it was introduced due to lack of options, the distinguishing between the website interaction and the scene interaction, can be seend as an usability improvement. It prevents the user from accidentally interacting with a website while trying to move an objects and vice versa.

But this solution raises the question, on how to implement website interaction in VR. In VR there is no mouse, let alone mouse events. Only the controllers and their input are accessible, but the input does not directly communicate with the browser and trigger pointer events. It has to be interpreted by JavaScript. Which lastly means the mouse events have to be triggered manually. As explained before, this is impossible, when dealing with iframes, which do not have the same origin as the site your code is running on. There are little to none ways to work around this. Further ideas and approaches on how to tackle this issue, can be read in section 3.6.

3.5 Interacting with others

The interaction with the local scene is important, but the actual idea is to connect multiple people in one room, thus we also need a interaction in between the clients. For that purpose the already existing ubi-interact backend described in section 2.4, is used. Its topic subscription functionality perfectly fits our needs. It allows for subscriptions for custom topics, which will be created if not already existing. So we a topic can be created for each room and each client can subscribe to them. To distinguish the rooms from each other, a so called room id got introduced. When accessing the base XRHub URL a new room id gets generated and a new topic is created on the server. When adding a room id to the end of the URL, such a subscription to the room topic with the given id, is initialized. Every client only subscribes to one room and the rooms are com-

pletely separated from each other. No room can retrieve information from another room.

In order to invite people to the room, a link on the headerbar has been added, which contains the XRHub URL with your own room id attached to its end. That way an information transfer between clients has been established. Now is the question what should be transferred? The first idea, was to create an architecture leaned on facebooks Flux architecture (subsection 1.4.5). This means every interaction from the user, does not lead directly to the user changing the scene. Instead an action creator is called, which first interacts with the server and upon successful transaction, created and dispatches an according action. The scene itself listens to actions and changes the scene depending on the action. For example when a user moves an object. The objects does not get moved immediately. First the moveObject action creator is called, which sends an update to the server, containing the object id, the action type and the action parameters, so the direction the object gets moved to. Since the messages are published via WebSocket, there is no response to wait on. Thus the action creator can continue with creating and dispatching a move object action. The scene then acts as the XRHub equivalent to the flux model store. It accepts the action and moves the object accordingly. The format we send the updates to the server is string, or more specific a string in JSON format. The format is explained in subsection 1.5.1. The ubi-interact framework is very customizable and the ubi-message-formats repository could be easily extended by custom actions. But since it wasn't yet determined what such an action contains, JSON was the format of choice. This keeps the application very flexible regarding the information sent and received. This architecture works really well when working with servers using REST, since it can be assured, that an action got approved by the server before it gets applied it to the local state. It helps keeping the server and the client state the same. But using WebSocket, makes the advantages of the architectural idea obsolete, since the server does not reply to the WebSocket messages. Thus the application cannot know if message has been received by the server. Another disadvantage comes with that architecture. Actions only define the transition between the previous state and the future state. So upon application of an action it has to be assured that the previous state from receiving client is the same as the one from the sending client. Since there is no persistency layer, this cannot be done. The room is instantiated locally. When a user joins a room, that already has been modified, he still starts with the base room. The then received updates are applied to the base state of the room, which is completely different from the one sending the updates. This leads to wrong outcomes, on both ends. One solution to that problem, is to store all actions applied since the creation of a room. Those can be then applied to the room of any joining user. This solution leads to an immense amount of data, that has to be stored on the server, for each room currently active. Also it can lead to a really high loading

time. If a user joins after an hour and the other participants were constantly modifying the room, he has to wait until all those actions get applied to the local room, which can take multiple minutes.

Those reasons lead to the introduction of a different update mechanism. Instead of using Flux the changes to the scene are directly applied and afterwards the whole object that has been modified get sent to the server. Opposed to the transition between states, the information now defines the final state of the object. This means it can be applied to any other client, without having to compare previous states. For example if a user joins late, the objects that get modified afterwards, will have the same positions for everyone. It's not a perfect solution since objects that do not get touched after a user joins will not be updated for that user and new users can overwrite previous work, by modifying such inconsistent objects on their own client. Still a better solution than the Flux one. But the persistency is not the only issue regarding the updates.

Since every client has its own instance of three.js, because the room gets instantiated locally, the object ids are different for each client. This makes it impossible to match updates from one client to the scene of another client. To solve that custom object ids have been introduced, that are stored in the userData property of the object. Those are set by the application and not three.js and follow a pattern, so they stay the same for each client. For example a website object has the id composed by the prefix "WebsiteCanvas" and the URL it is currently showing. This helps to identify objects cross client, but comes with its downsides. With this logic, website canvases showing the same website have the same id. Additionally by generating the ids on each client side there are multiple sources for the same information, which is not recommended. When working with data, you always should have a single source of truth, to be sure that the data stays consistent. So what is needed, is a storage on the server, which stores the scene state with its objects and object ids.

The ubi-interact-backend already comes with a storage functionality, which can be used for that purpose. The base storage can be inherited and its methods can be modified to fit the requirements for a room storage. The question i: How do the rooms get stored? The storage itself only manages the information. The type of information has to be defined by ourselves. The JSON format described in subsection 1.5.1, was used as serialization. JSON was the winner because, even though other formats as gLTF or glb are more efficient and standards for storing 3D environments, they are difficult to modify after serialization. But exactly this is what is needed, since the server side also has to update a scene, in order to stay consostant. The server only has access to the serialized form though. JSON is also very well supported by three.js. Each

Object3D in three.js, including top level classes, like scenes, have a *toJSON()* function, which immediately serializes them to the JSON format with a three.js specific scheme. Furthermore three offers an ObjectLoader which can transform JSON objects back into Object3D objects. This makes it possible to easily im- and export three.js scenes and objects. One more big advantage, is the fact, that when using the three.js scheme, the uuids of the object also get serialized and deserialized. This eliminates the need for custom object ids.

Since each room has two scenes, the WebGL scene and the CSS3D scene, an object with two scene properties and a roomId, to identify the room, has to be stored. The scene properties contains the exported JSON content as a string. This object is delivered every time a client requests a room. Rooms can be requested by a GET service. The request is sent with the room id at the end of the room get service URL. If the server has a room stored with the given id, the room is returned, otherwise a base room object gets copied, which is always present. It gets saved into a new room with a new id and then sent back to the client. Which means the client always receives a valid room. If the room id received from the server differs from the requested room id, the client room id will be overwritten. The advantage of this setup is, that every client now has a single source of truth, which is the server. This secures, that the ids of the objects are the same for every client and that the updates sent back and forth can be applied.

Also important is how the way the updates are sent has been modified. A update can be triggered by every mouse move event, which happens every few milliseconds. Sending data that often can easily overwhelm the connection or even the servers or clients themselves. This can lead to crashes and data loss. To prevent that the updates are not sent every time a mouse event occurs, instead a map with the updates, using the object uuid as the key, is filled. So if an update for that object already exists it gets overwritten by the new one. Additionally the JavaScript function *setInterval()* has beed used, which keeps executing a given callback in a certain time interval. The time interval has been set to 800 milliseconds and a callback, that iterates through the map pass a and sends the corresponding updates, has been attached. After being sent, the updates get deleted from the map. This way updates are only sent once every 800 milliseconds and only once per object. If an object has been moved multiple times in that time frame, does not affect the outcome, since the map contains its final position.

In order to persist a rooms state even after reloading the site, the server needs to keep an updated state in the storage. This happens by incrementally updating the saved room, when receiving an update. The server then takes the specified room and searches for the object that has been updated. The existing instance of that object in the

room, is then replaced with the updated version. Finally the room is saved again. This would not be possible with a gLTF or glb serialization, since glb is stored in binary form and really costly to be deserilazed on the server. gLTF and glb store no ids for the objects, which leads to the problem of having to identify objects across all clients. Both also have a way more complex structure than the JSON scheme from three.js. That's why JSON won.

That settles it for the distribution of the user interactions regarding the scene objects. But what about the interaction with the website itself. Everybody should be able to see the same thing, so the websites also need to be synchronized. This can be done in various ways. For example by sending each MouseEvent, that occurs on an iframe and applying it to the iframe in the other clients. This is easy to implement since the ubii already supports a message format for MouseEvents. Another solution would be to have only one client being able to interact with the site and the others only get an image of that site. Both of these solutions work fine, when working with same origin websites, but the moment you embed a foreign website, things become way more difficult to realize. More on why those approaches do not work and how you could still solve it, can be read in section 3.6.

Another important thing to mention is the way the scene is deserialized in the client. Since the iframes cannot be serialized with the toJSON method from three.js, the scenes deserialzed by three.js do not contain them. Even more information is missing. For example the reference from a WebGL plane, to its CSS3D counterpart is not present anymore. All this information has to be redefined. This is done, by instantiating a ThreeWebsiteCanvas with the already existing objects passed to the constructor. But how can the WebGL and CSS3D object pairs be extracted the from the two scenes? By introducing a canvas id, which is unique for each ThreeWebsiteCanvas. Both parts that build such a canvas object have the same canvas id, which makes it possible to identify the wanted pairs. The contructor receving the two pobject, now has to fix them. The WebGL part is straight forward, since only the reference to the CSS3D object has to be redefined. The CSS3D part is not that easy. The problem is, that after creating an CSS3D object, the once inserted HTML element cannot be replaced. Since the deserialization from thee.js could not deserialize the iframe, the received CSS3D object has no HTML element attached to it. Which means it does not display anything. This can be solved, by instantiating a new CSS3D object, with a new iframe with the same url. When complete, the old object is swapped with the new one and the scene renders as expected.

3.6 Limitations to iframes

The common way to embed other websites into your own is by using iframes. There are other ways, like using the <*object*> or <*embed*> tag, but the difference between those three is small. Formerly iframes were considered having security flaws, since the same origin policy was not that established yet and there was no way for the browser to check if the iframe origin is secure or not. Nowadays with HTML5, those concerns no longer apply and iframes are as secure as the other two options. No matter which way you embed the website, you'll always encounter the same drawbacks.

Most websites only allow specific parts of them to be embedded, for example YouTube. The website itself prohibits being embedded, by using the X-Frame-Options HTTP-header and setting its value to "DENY". Only the videos themselves can be embedded. This already limits the amount of websites that can be embedded and forces the way they have to embedded. But this is only one of the "issues" regarding iframes. The main problem is the strict same-origin policy described in subsection 1.4.3.

This policy makes it impossible for websites to interact with each other on one client. Of course it has its reasons to do so, since not having this limitations would open endless possibilities for malicious attacks to take place. But in our case it mostly just hinders the implementation of 2D interaction in a VR environment. As explained in section 3.4, a way to trigger MouseEvents manually is needed, in order to map the user input to events the website can react to. Same goes for the issue described at the end of section 3.5. This is impossible due to the fact, that the DOM elements of a cross origin iframe cannot be accessed. But those elements are needed to trigger the events on them. Meaning thanks to the same origin policy neither the VR interaction with websites, nor the website synchronization across all clients can be implemented. To be fair you are not completely locked out of the iframe. There are still ways to communicate. The HTML5 feature *postMessage* can be used to send and receive messages across origin borders. The problem is that this feature also has to be implemented by the site that is embedded, which for most sites is not the case. And even if so, you are limited to what has been implemented by the third party which can be completely different, for each website you access. There is only one scenario where this feature comes in handy and this is when embedding videos or streams. Most media platforms like twitch.tv or youtube.com have implemented a *poseMessage* API. And is is very similar across those platforms. The API allows for videos to be played an paused and in some cases even the volume is adjustable and the video can be skipped. So at least videos could be made interactive. But it would need you to know when a video is displayed and when not and adjust the messages sent according to the site.

The same origin policy also denies the second solution for the syncing, which is sending the image of the website. To see why you need to understand how screenshots are taken in browsers via JavaScript. There are multiple frameworks simplifying the act of taking screenshots, like html2canvas (https://html2canvas.hertzen.com/), but all of them work the same way. They use the HTML <*canvas*> element. It is used to draw graphics via JavaScript and has multiple methods for drawing, including a method to draw the content from other HTML elements. So any HTML element together with a position you want to draw it at, can be passed into the *drawImage* method. The canvas then iterates through the element and all its children and redraws their current content onto itself. Afterwards the canvas contains a snapshot of the given HTML element. Next it can be transformed into png via its *toDataURL()* method. This method converts the image inside the canvas into the desired format and returns a data URL, which either can be embedded into an <*image*> element, or used to download the image. The standard format returned by *toDataURL()* is png. This seems perfect for us to use, since we can simply pass the iframe we want a screenshot of to the canvas and do not even have to worry about angles or distances, because the canvas will always draw the iframe in 2D as if it was displayed directly on the page. But here the same-origin policy comes into play again. Since the canvas needs to iterate through the HTML element in order to draw it, it needs access to the element and its children. This access is denied if the canvas is from a different origin than the element it wants to draw, thus making it impossible for the canvas to draw the content. So not even screenshots are allowed, because they could also be exploited and used as an attack vector.

Taking a screenshot directly via the three.js renderers, also has been attempted. The WebGL renderer works with a canvas element and uses the WebGL library to draw onto it. So its DOM element already is a canvas, which can be directly converted into a png. Sadly the iframe is in the CSS3D scene not the WebGL scene. The CSS3DRenderer works completely different and does not actually render anything itself. Its only responsibility is to place the objects in the given scene correctly. Which means it transforms all objects into camera space. Since each object consists of plain HTML the actual rendering is done by the browser, leaving us with the same problem from before.

There's one more way, to get the website interaction realized and even enable screenshots. The browser can be tricked into thinking, that embedded site is from the same origin. This can either be done, by a Proxy server manipulating the headers of the request, or by manually manipulating the HTML. The first idea is to use a Proxy server, which acts as man in the middle between you and the desired site. Instead of requesting the URL directly from the client, it is encoded and a request is sent

containing the encoded URL to the Proxy server. So the *src* attribute of the iframe is set to the Proxy request with the encoded URL. The URL gets decoded from the server, which then sends a request to the actual source. Upon receiving a response, the Proxy takes the response and manipulates its headers. It sets the value of the *Access-Control-Allow-Origin* and *X-Frame-Options* attributes to the origin of the actual requester. Finally the manipulated response is sent to the client. By doing this, the browser thinks, the received website is okay with sharing resources and stops enforcing the same-origin policy limitations. This is also part of the way mozilla hubs has realized their video embedding feature. It has more to it, but they also are using a Proxy.

The second way is to not even use an iframe. Instead the URL is directly requested in your code. The received response is then stored in a <*div*> element as its *innerHTML* attribute. This div can then be inserted into a CSS3D object. Now the browser does not even know that it contains content from a different site, because the content directly has been added directly, without embedding it. But there is a problem with this approach. All references to images, scripts and links, that are relative to the actual origin are now broken. The browser tries to get them from the parents origin, which, in this case, is our client, but they do not exist at that location. So the site gets rendered, but most of it is not displayed correctly. In order to solve this the div element created earlier on is taken and a recursive iteration through its children is started. When encountering a reference that does not start with "http", which means it is a relative reference, the requested URL gets added in front of the original reference. This way the browser requests the resources from their actual origin. Now the site looks close to normal, but it is still does not look as when importing it with an iframe. This is because the CSS styles of the client overwrite some of the styles of the embedded website. Since the website is now treated as a child and not as a distinct website, it inherits the styles from the parent, leading to black backgrounds instead of white ones etc. This could be fixed by manipulating the styles of the website and setting the *!important* tag on all of their values, which makes them overwrite the parents styles. But most websites do their styling via a separate file, which means the file name containing those style has to be found out. Then it has to be manually requested from the website. Next its values have to be adjusted and finally the styling element in the div element has to be found, and the original file reference has to be swapped out by a dataURL reference to the modified styles. After doing that, most of the site should look as expected.

Both approaches are not recommended, since they bypass the built in browser security measures. Other sites can exploit this, by extracting information from our page or manipulating its content. And users can abuse the fact that a proxy is implemented and use it as a gateway to bypass country restrictions, or do malicious things to others.

Due to the lack of remaining options, one of those methods has to be realized. If you have to chose between them, the better solution would be the proxy. First of all there is no need to go through the whole HTML document. Secondly the chances of the website looking and functioning as expected are way higher. But no matter which way you go, it is important to implement security measures by yourself. An option would be to keep white a white list containing all allowed websites. You only embed the site, if it is on that list. This is not that user friendly, since if a user wants to embed a site not already known, he first has to request that it gets added to the white list, but it would assure that all sites accessed by the user are secure.

In conclusion iframes are a good way to import foreign websites, as long as no interaction via JavaScript is needed. If that is the case, the only option to still realize that wanted behaviour, is to trick the browser. This comes with loss of security, because the actions of the embedded site cannot be controlled. On the other hand it makes it possible to develop a better VR experience. Since by bypassing the browser security, all same-origin policy restrictions are no longer apllied. Meaning all methods mentioned in section 3.4 and section 3.5 for implementing VR interactions and synchronizing the users clients, can be applied.

4 Future Work

This chapter contains ideas for further improvements, that can be implemented in the future.

4.1 Tackling the CORS issue

One of the first things that should be worked on, is the embedding of foreign websites. As explained in section 3.6 the interaction with iframes is strongly limited, which finally lead to the project not being finished completely. But there are ways to work around it. The solutions from section 3.6 could be implemented or even a better one could be developed. Other solutions could be taken into account, like the one mozilla hubs used for their video embedding feature.

4.2 Implementing VR interaction

Even though the XRHub already supports VR, the current state of the WebXR API makes it impossible to use the functionality. While this can't be changed. You can wait for WebXR to stabilize and then try to get a running combination of three.js and browser versions. Then you could start implementing the VR interaction in XRHub. Most of the code should be applicable from the mouse and keyboard interaction already implemented, since it was implemented to be as close to VR as possible. Furthermore you could start making usability adjustments. Of course usability was a big factor in this thesis, but since we lacked the possibility to try things out or even implement them, there is no real idea of how the introduced interaction concepts translate to VR. It would be interesting to know which interactions can be improved and how so.

Additionally you could implement interactions for multiple types of XR hardware. E.g. the *Microsoft HoloLens* or just smartphones. Especially smartphones would increase the user base by a lot, since they are way more accessible than actual XR hardware. It would also mean a completely different interaction approach, since you have different types of input. While the Holo Lens has gestures instead of controllers, smartphones only have a gyroscope.

Smartphones also could be used in a different way. You could try to combine the XRHub with former ubi-interact projects, like the textitVR Keyboard or the *VR Laser Pointer*. This would prdouce even better user experience. The *VR Keyboard* for example, could facilitate website interaction a lot. Typing in VR is quite difficult and time consuming, which makes a lot of people rather take off their headsets and use the keyboard. This constant switch between the real world and the virtual reality can be very exhausting. With the smartphone feature, users could change the URL of a website, login or fill in a form, without having the need to exit the VR experience.

4.3 Introducing new features

The base idea of XRHub already is a viable product which introduces new ways of communication, but it can be extended. There are multiple other features, that would improve the user experience, when using the XRHub. Mozilla hubs for example gives a lot of inspiration on what to implement next.

Things like video communication would be helpful. By being able to share your computer screen or the input of your smartphone camera, you can demonstrate things you otherwise would have to show via third party software on a 2D interface. It also enables you to redesign your workplace inside VR. For example you could implement a browser extension, that enables you to split your screen onto a browser window. With this possibility you could bypass physical limitation, like only having space for one monitor. You could use a VR headset and create a setup with 3 or more monitors inside XRHub, with which you could work instead. While the feature of splitting your desktop onto browser windows is a more futuristic idea. Just implementing the ability to share your screen, would already enable you to visualize way more things at once while working on something. Instead of having to tab out to search something, you could just open a new website in the XRHub and place it next to your desktop sharing plane. And this are only the possibilities without the interaction between two users. The collaboration woth such a feature could be way easier and more efficient. Furthermore you would not be limited to only one of them sharing the screen, like it is in most 2D communication tools. Everybody could do so.

Voice and text communication is also something that should be considered for the future. To understand the context of the visible information it can be very helpful to have an audio explanation of what is going on. So a person being able to explain what is being displayed would help a lot. Additionally it would enable the users to have a

discussion. Text would also find its use and if it only was for copying code from one client to another, it would erase the need to write things off.

All the above features make sense to be introduced to XRHub. But even though they are very valuable, the priority to implement them does not necessarily have to be that high. When implemented correctly, the base version of the XRHub could already be able to do realize most of the scenarios described above. It could use the internet to its advantage. Since in the best case, any website is reachable from within a XRHub room, you could use already established communication tools like glsdiscord or Google Hangouts to do the communication for us. Instead of having to implement a Desktop sharing feature, you just have client specific instances of websites. Those are not shared to other rooms, but they still can be used as a gateway to the others. For example, by logging into your glsdiscord account and streaming your desktop in there. But it doesn't stop there, multiple other tools already existing in the world wide web could be used. For example draw.io or other drawing websites could be used instead of having to implement drawing to the XRHub itself. Of course drawing in the 3D space would have more advantages, since you also could draw outside of the websites frame, but it would fullfill most needs. And that is where the true power of the XRHub lies. It is not limited to its own features, it can extend and evolve in any imaginable way, as long as there is a website for it.

Another future extension could be to introduce avatars. A lot of XR hardware has the capabilities to track the users body. This information could be used to display their bodies inside the room as an avatar. It would drastically increase the immersion and make the whole interaction between the users more personal. Users could support their vocal expressions with gestures and point at specific things in the room. It would also enable them to interact with the room without using the controllers. You could make those avatars collide with other objects in the scene. This way a user could push things and objects would bounce off of them, which makes the experience as a whole feel more realistic and engaging.

Another point on the list is implementing a room creation tool. A tool, where the user can insert objects and place them all around the room. First you could offer a selection of predefined objects, but later on you could introduce the ability to import objects and add those to the room. This could be realized with the GLTFImporter already implemented by three.js. Since most 3D objects can be described in the gLTF format, it would allow for an immense amount of options when designing a room. But why should it stop at objects? You could implement an import for whole rooms or other types of 3D scenes. One very attractive feature, would be the ability to import scripted

rooms, like games or simulations. The addition of a web browser in VR games or simulations could change the way they are played or used. For example scientist could have a running simulation of an artificial intelligence or a physical experiment and watch it from close up. While simultaneously being able to open a website containing the specifics about the simulations, like for example the state of the neural network, or specific values about the pressure or force generated in the physical simulation. Without being limited to one person only, multiple scientist could join that room and experience the events together, while discussing and elaborating the results. It is very helpful to be in there together, since it is hard to explain a VR experience to others, without them having had the same experience. Those extension also come with downsides and dangers that have to be thought of. For example enabling others to insert their own scripts is a big security issue, since they can run any type of JavaScript. This could be abused to spy on others or exploit the system in some other way.

Opposed to the approach of trying to bring more features into the XRHub, you also could think in the other direction and see the XRHub as a feature for others. It could be designed as a module, that can enable 2D interfaces in VR. This module then could be used by anyone developing in VR to extend their own projects. It would open up way more possibilities for others to design and implement their ideas.

4.4 A future with mozilla hubs

As explained in section 2.3 mozilla hubs already comes with a lot of features, that would fit perfectly to the XRHub. So instead of reimplementing all those features by ourselves, one could use the already existing platform and its community and realize the XRHub there. Since mozilla hubs is an open source project, there is no barrier preventing us from extending it. When a good solution for the 2D interaction in VR has been found, it is a valid option to try to implement it inside of mozilla hubs. Depending on the solution this can be impossible. But if realizable, it would immediately upgrade the XRHub with a lot of features. This comes with the downside of being stuck with the technology and design decision, that have been made by the mozilla hubs developers. For example a mozilla hubs room does not include any scriptable objects. This means it is not designed for games or simulations, where objects have logic applied to them. This reduces the amount of use cases you could realize with the mozilla hub/XRHub fusion. Nonetheless it should be looked into.

4.5 Testing

One major thing the XRHub is lacking are tests. It has not been built test driven. This means most of the logic isn't tested at all. Test are very important when continuously developing a software. There are multiple types of tests in software development. Unit tests for example, try to split the code in as small parts as possible and test the logic of these parts separately. They do not just assure, that the code logic does not get altered, they also can serve as on board documentation for people that have not worked with the code. Since they are designed to be as small and simple as possible, they mostly are easy to understand and help to clarify how certain code parts work. Opposed to unit tests, end to end (E2E) tests try to test the application as a whole. Those tests, also known as black box tests, take the built application and check if it performs as expected in the given scenarios. One benefit both types of tests have, is that they can be executed automatically. This helps to counteract quality regression, which is a common issue in software development. You cannot completely prevent your software from having bugs, but you can fix a lot of them to improve your software quality. Now when implementing new features, new bugs can be introduced and even old ones can find its way back into the code. This is where the tests come in. They test the application on a regular basis and make sure everything works as intended. Whenever you find a bug and fix it, you can write a test case testing the scenario that lead to the bug, so in the future you will immediately know when a bug reappears. XRHub lacks both test types. While unit tests are easy to append, the implementation of automated E2E test would bring a real challenge. One would have to implement those tests for a all the supported devices, which on itself already is a demanding task. But most importantly you would have to develop E2E tests for virtual reality environments. There is no good solution yet, to test web-based VR interactions, thus XRHub would fit very well as platform, to elaborate possible methods. It eventually could be realized, by extending the existing WebXR Device API emulator from Mozilla, in order to be able to generate WebXR interactions via code. This could then be used to simulate scenarios in VR. the assertion of the outcome could be done by checking the position of the objects in the room, comparing the rendered images to each a expected image, or by accessing the DOM elements directly.

5 Conclusion

Websites in VR are not easy to realize. They already exist via implementations like the Oculus Browser, but those are only usable locally. Trying to implement such a feature in the web environment, leads to a lot more issues. The main problem is security. The same-origin policy of browsers prevents the interaction between two cross origin websites, thus eliminating all options to produce a VR application with interactive websites. It can be bypassed by tricking the browser via reverse proxies, but it the policy is not implemented for fun. Bypassing the security feature means a lot of danger, since all JavaScript code can now simply cross origin borders and spy on or modify other websites. This is not just an issue for the XRHub itself, it also could enable attacks on other sites opened in XRHub. With all those downsides this risks still seems to be worth taking. The ability to visit any website within a VR experience, opens the door to a new dimension of possibilities. It welcomes all web technologies into the VR world. An endless amount of combinations between web technologies and 3D worlds can lead to completely new ways of experiencing VR and the web. Combined with the ability to have this experience from any device that has a browser, without having to download or install anything, the XRHub would probably enable a lot of people, to do things they would not be able to do otherwise. So websites in VR are not just possible, they also seem to be worth the effort. Now they only need to be implemented.

List of Figures

Bibliography

[1] URL: https://developers.google.com/protocol-buffers (visited on 08/12/2020).

[2] URL: https://gitlab.lrz.de/IN-FAR/Ubi-Interact/ubii-msg-formats/-/blob/develop/src/proto/topicData/topicDataRecord/dataStructure/vector3.proto (visited on 08/12/2020).

[3] *CSS Snapshot 2018 W3C: Working Group Note, 22 January 2019*. URL: https://www.w3.org/TR/CSS/#css (visited on 08/09/2020).

[4] A. Etienne and J. Etienne. *Mixing HTML Pages Inside Your WebGL*. 2013. URL: http://learningthreejs.com/blog/2013/04/30/closing-the-gap-between-html-and-webgl/ (visited on 08/07/2020).

[5] *Hubs Features*. URL: https://hubs.mozilla.com/docs/hubs-features.html (visited on 08/11/2020).

[6] A. M. I. Fette. *The WebSocket Protocol*. URL: https://www.hjp.at/doc/rfc/rfc6455.html (visited on 08/11/2020).

[7] *In-Depth Overview*. URL: https://facebook.github.io/flux/docs/in-depth-overview (visited on 08/09/2020).

[8] *Introduction*. URL: https://aframe.io/docs/1.0.0/introduction/ (visited on 08/11/2020).

[9] *Introduction - What is Vue?* URL: https://vuejs.org/v2/guide/ (visited on 08/09/2020).

[10] P. W. Johannes Ebke. *The A4 project: physics data processing using the Google protocol buffer library*. URL: https://iopscience.iop.org/article/10.1088/1742-6596/396/2/022012/pdf (visited on 08/12/2020).

[11] C. Labs. *Article-World, View and Projection Transformation Matrices*. URL: http://www.codinglabs.net/article_world_view_projection_matrix.aspx (visited on 08/07/2020).

[12] *MouseEvent*. URL: https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent (visited on 08/11/2020).

[13] M. G. Nell Waliczek Brandon Jones. *WebXR Device API - W3C Working Draft, 24 July 2020*. URL: https://www.w3.org/TR/2020/WD-webxr-20200724/ (visited on 08/11/2020).

[14] *r112*. URL: https://github.com/mrdoob/three.js/releases/tag/r112 (visited on 08/11/2020).

[15] *Same-origin-policy*. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (visited on 08/09/2020).

[16] *Standard ECMA-404 The JSON Data Interchange Syntax*. URL: https://ecma-international.org/publications/standards/Ecma-404.htm (visited on 08/12/2020).

[17] *The JSON Data Interchange Syntax*. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on 08/12/2020).

[18] N. H. Thomas Akenine-Möller Eric Haines. *Real-Time Rendering Third Edition*. Natick, Massachusetts: A K Peters, Ltd., 2008.

[19] *Three.js Fundamentals*. URL: https://threejsfundamentals.org/threejs/lessons/threejs-fundamentals.html (visited on 08/07/2020).

[20] *Ubi-interact*. URL: https://wiki.tum.de/pages/viewpage.action?pageId=71313835 (visited on 08/13/2020).

[21] *WebGL Overview*. URL: https://www.khronos.org/webgl/ (visited on 08/07/2020).

[22] *WebVR Editor's Draft, 12 December 2017*. URL: https://immersive-web.github.io/webvr/spec/1.1/ (visited on 08/11/2020).

[23] *Welcome to Hubs*. URL: https://hubs.mozilla.com/docs/welcome.html (visited on 08/07/2020).