communicated by:
Lehr- und Forschungsgebiet Informatik 9

Prof Dr. Ulrik Schroeder

The present work was submitted to the Learning Technologies Research Group
Diese Arbeit wurde vorgelegt am Lehr- und Forschungsgebiet Lerntechnologien

Realization of the OmiLAXR Framework in WebXR Environments
Realisierung des OmiLAXR Frameworks in WebXR Umgebungen

Bachelor-Thesis
Bachelorarbeit

presented by / von

Bekyarov, Alben
378220

Examined by / Begutachtet von

Prof. Dr.-Ing. Ulrik Schroeder
Prof. Dr. rer. nat. Horst Lichter

Supervised by / Betreut von

Sergej Görzen M.Sc. RWTH

Aachen, April 7, 2025

Contents

List of Figures

List of Tables

List of Listings

Abstract

In recent years, there has been growing interest in using Extended Reality (XR) in education. Virtual learning environments can be useful not only in scenarios where hands-on training is potentially dangerous, expensive or simply difficult to organize in real life, but also for simplifying complex topics through visual elements or boosting learners' motivation by making the experience more engaging. However, as more XR-based educational tools are developed, one challenge becomes clear - how do we actually measure and understand what learners are doing in these virtual environments?
The OmiLAXR framework was developed as a potential solution to that problem. Essentially, it is a modular toolkit designed to simplify the tracking of learners' actions and behaviors in virtual environments by providing a simple way to integrate Learning Analytics in any XR-based application. It does this by monitoring user interactions and generating structured, meaningful data about the detected experiences. For that, OmiLAXR relies on the Experience API, thereby helping to solve a second problem - data standardization. By using a consistent data format for the collected information, it enables the comparison of results across different systems or institutions.
This thesis explores how OmiLAXR's functionality and design can be realized in the WebXR ecosystem. Ultimately, two approaches were tested. The first one, which combined C#, WebAssembly, and JavaScript, aimed to preserve the original architecture but ultimately did not align well with the framework's core principles. Then, the second one, built almost entirely in TypeScript, yielded better results and was therefore explored in more depth.
The result is a functional, browser-based version of the OmiLAXR framework that can be integrated into different WebXR applications with minimal setup. While there is still room for improvement, the work presented by this thesis proves that creating a robust Learning Analytics framework like OmiLAXR for the web is not only possible but practical.

Kurzfassung

In den letzten Jahren ist das Interesse an der Nutzung von Extended Reality (XR) in der Bildung stetig gewachsen. Virtuelle Lernumgebungen können nicht nur in Szenarien nützlich sein, in denen praktisches Training potenziell gefährlich, teuer oder einfach schwierig zu organisieren ist, sondern auch dabei helfen, komplexe Themen durch visuelle Elemente zu vereinfachen oder die Motivation der Lernenden zu steigern, indem sie das Lernen spannender macht. Mit der Entwicklung immer mehr XR-basierter Lernapplikationen wird jedoch eine zentrale Herausforderung deutlich – wie können wir eigentlich messen und verstehen, was Lernende in diesen virtuellen Umgebungen tun?

Das OmiLAXR-Framework wurde als mögliche Lösung für genau dieses Problem entwickelt. Im Wesentlichen handelt es sich dabei um ein modulares Toolkit, das die Erfassung von Aktivitäten bzw. Verhalten in virtuellen Umgebungen vereinfachen soll, indem es eine einfache Möglichkeit bietet, Learning Analytics in beliebige XR-Applikationen zu integrieren. Dazu überwacht das Framework die Interaktionen der Nutzer und generiert strukturierte, aussagekräftige Daten über die erkannten Erfahrungen. OmiLAXR verwendet dafür die Experience API (xAPI) und trägt somit zur Lösung eines zweiten Problems bei – der Standardisierung von Daten. Durch die Verwendung eines konsistenten Datenformats für die erfassten Informationen wird der Vergleich von Ergebnissen zwischen verschiedenen Systemen oder Institutionen ermöglicht.

Diese Arbeit untersucht, wie sich die Funktionalität und die Struktur von OmiLAXR im WebXR-Ökosystem umsetzen lassen. Insgesamt wurden zwei Ansätze getestet. Der erste, der C#, WebAssembly und JavaScript kombinierte, hatte das Ziel, die ursprüngliche Architektur beizubehalten, erwies sich jedoch letztlich als nicht gut geeignet im Hinblick auf die Kernprinzipien des Frameworks. Der zweite Ansatz, der fast vollständig in TypeScript umgesetzt wurde, lieferte bessere Ergebnisse und wurde daher eingehender untersucht.

Das Ergebnis ist eine funktionale, browserbasierte Version des OmiLAXR-Frameworks, die mit minimalem Aufwand in verschiedene WebXR-Applikationen integriert werden kann. Auch wenn noch Verbesserungsmöglichkeiten bestehen, zeigt diese Arbeit, dass die Umsetzung eines robusten Learning-Analytics-Frameworks wie OmiLAXR für das Web nicht nur möglich, sondern auch sinnvoll ist.

Chapter 1 Introduction

In the last few years, especially with challenges like the COVID-19 pandemic, we have seen how important improving education is [Ver24, ACISIAA22]. One way to do this is through the use of technology [Ver24, ACISIAA22]. An increasingly popular idea towards that goal is to incorporate XR into our teaching and learning practices [ACISIAA22, GHS24b]. Leveraging XR technologies in education certainly has the potential to enhance the learning experience [ACISIAA22]. It could, for example, be useful in dangerous scenarios like flight simulations [RG23] and surgical practices [TTFA24], in distance learning [WCD+24] and many more. Furthermore, XR-enhanced learning could also improve students' motivation as it makes the whole experience feel more engaging [Ded09, ACISIAA22]. However, while interest in XR-enhanced learning has grown, its practical adoption is still in the early stages [GGL21, ACISIAA22]. Incorporating the technology in education is not a simple task [GGL21]. There are many challenges that need to be solved first [GGL21, ACISIAA22]. One such challenge that needs to be considered as well is how we measure the effectiveness of these new learning applications [LC25]. Traditional performance metrics like quiz scores are not sufficiently expressive in this scenario where learning is experienced in a different way [LC25, ACISIAA22]. This is where the *OmiLAXR framework* comes in [GHS24b, GHS25]. It enables the integration of Learning Analytics in XR applications by providing developers with the ability to track a variety of different interactions and experiences within a virtual environment and to generate detailed information about them (e.g., in the form of xAPI statements). Moreover, the framework aims to address challenges such as the lack of standardization and the amount of manual work by reducing repetitive tasks, establishing a consistent standard for data collection that can be compared across different institutions or platforms, and more [GHS24b, GHS25, GHS24a].

Goals of the thesis

The goal of this thesis is to explore how OmiLAXR's core functionality can be realized in WebXR environments while also designing a solution that aligns with the framework's principles, such as being *developer-friendly* and *enabling some level of freedom* in how the project is used. Ultimately, we are motivated by the **research question** "*How can the OmiLAXR framework be realized in WebXR environments?*". However, another important aspect of adapting a framework like OmiLAXR for WebXR lies in the reach of the web platform. WebXR applications can be accessed through standard browsers without requiring installations or platform-specific dependencies, making them usable on a wide range of devices [MS18]. This allows us to offer the framework's functionality to a broad user base and various types of applications, thereby making the framework even more accessible. In our attempts to achieve

this goal, we are going to examine two different approaches - one using C# and another using TypeScript.
The paper is structured as follows: Chapter 2 introduces the necessary background knowledge. Chapter 3 presents the original OmiLAXR framework with its technical foundations. Chapter 4 explores how the framework can be adapted for WebXR applications by reviewing two different approaches. Chapter 5 examines potential improvements of the project presented in Chapter 4. Finally, Chapter 6 concludes the thesis.

Chapter 2 Background

Before we explore the technical aspects of OmiLAXR or its realization in WebXR environments, it is important to first build a foundational understanding of the key concepts and technologies involved in this project. This chapter provides a brief overview of the topics relevant to this work. The goal is to equip the reader with the necessary background to better understand the motivation and design decisions discussed in the following chapters.

2.1 Learning Analytics

General definition of Learning Analytics

A general definition or model for Learning Analytics (LA) is provided in the following papers: [Sie13, HGD+24, CPL20, CDST12]. Essentially, it could be defined as "systematically gathering and analyzing learner data to make the learning process more efficient or more effective" [Sie13, HGD+24]. LA focuses on different goals, such as identifying strengths and weaknesses of the students, giving feedback on the learner's performance, etc. [Sie13, SA24]. As the adoption of XR technology in education grows, the possibilities for collecting information on how learners progress through the course are also growing [SSCD23, SA24, CPL20, HGD+24]. Virtual learning environments are an excellent space for gathering data [SA24, HGD+24, CPL20]. Moreover, XR-based learning applications allow for the collection of a wide variety of data beyond simple statistics that only show whether a student has passed the course or not [BW16, SA24, HGD+24]. Here, we are dealing with multimodal data (e.g., eye-tracking, gestures, moving around the virtual space, etc.) which offers deeper insights into the learners' behaviors, trends, etc. [BW16, SA24, HGD+24, CPL20]. For example, in a medical VR simulation, LA could be used to track whether a student follows correct procedural steps [SDS22, TTFA24, SA24], while in a flight simulator, we could analyze reaction times and decision-making [RG23, SA24]. Such information can also be used in different domains (e.g., educators can identify exactly where students struggle, developers can evaluate their learning applications, etc.) [SA24, SSCD23].

Despite its advantages, Learning Analytics also comes with some challenges. The most obvious one is that personal information must be handled in a safe and ethical manner [CE23, CPL20]. Then, there is the problem of how much we can rely on that data and how exactly it should be interpreted (e.g., spending less time on a lesson does not necessarily mean a student did not understand it) [CE23]. Furthermore, since we are not dealing with simple information but rather complex multimodal data, some level of standardization of that data is required [CE23].

2.2 The Experience API

The Experience API[1] (or xAPI for short) is a standard for the collection of data about the experience a learner can have in a (possibly virtual) learning environment (see Figure 2.1 which showcases "xAPI's learner-centered approach") [KR16, HEGS22, GHS24b]. The project was started by the Advanced Distributed Learning Initiative (ADL) and later continued by Rustici Software, calling the project Tin Can[2]. In contrast to usual, well-known APIs that, for example, enable certain func-

Figure 2.1: An illustration of xAPI's learner-centered approach, gathering data from different sources and tracking multiple learning activities at the same time.
Image origin: `https://xapi.com/overview/`, Accessed on March 1, 2025

tionalities or services through HTTP requests or programming libraries, xAPI should rather be seen as a tool that provides a clear structure for data collection (in a learning/teaching scenario) [KR16, HEGS22]. It defines how the data we are dealing with should be formatted [KR16, HEGS22]. Furthermore, the TinCan library (the xAPI library) provides a way to send and store the gathered data into specialized databases called *Learning Record Stores* [KR16].
Readers might be familiar with SCORM[3] (Sharable Content Object Reference Model) [IMEH16]. SCORM has a similar concept to xAPI, but there are a few differences between the two as well [IMEH16, KR16]. Nevertheless, a problem that both technologies aim to solve is the issue of standardization so that, for example, results collected from different platforms, organizations or institutions can be compared [IMEH16].

Core concept of the Experience API - xAPI statements

The core concept of the Experience API are its statements [KR16, HEGS22]. Data collected in respect of the xAPI standard follows the *actor-verb-object* schema [KR16, HEGS22]. In simpler terms, we are describing a given interaction or experience in the "User performs a Verb on an Object" format (see Figure 2.2 for clarity). In certain cases, however, we might be interested in providing further details about the tracked interaction. For such occasions, xAPI has the concept of the *xAPI Extensions*[4]. They are used to insert additional details to the statements [KR16, HEGS22, GHS24b]. Another key feature of the Experience API are the IRIs (Internationalized Resource Identifier) or URLs (uniform resource locator) used in the statements [GHS24b, HEGS22]. In xAPI, the verbs, the objects and the extensions typically have URLs or IRIs that lead to a new component of the xAPI concept -

[1] `https://xapi.com/overview/`, Accessed on March 1, 2025
[2] `https://xapi.com/tin-can-evolution/`, Accessed on March 1, 2025
[3] `https://scorm.com/?utm_source=google&utm_medium=natural_search`, Accessed on March 1, 2025
[4] `https://xapi.com/blog/deep-dive-extensions/?utm_source=google&utm_medium=natural_search`, Accessed on March 1, 2025

Figure 2.2: Example of a basic xAPI Statement
Image origin: `https://xapi.com/statements-101/?utm_source=google&utm_medium=natural_search`, Accessed on March 1, 2025

the *xAPI Registry*[5] [GHS24b, HEGS22, EHL+20]. An xAPI Registry is like a database that holds the vocabulary (i.e. the verbs, objects, extensions, etc.) with their standard definitions and descriptions [GHS24b, HEGS22, EHL+20]. As standardization is important in the field of xAPI, the registries are a big part of the solution to that issue [EHL+20, HEGS22]. Without official registries different organizations may define the same verb or activity in different ways and use it in different contexts leading to inconsistencies across the data, making comparing results less effective (see Figure 2.3 for clarity) [HEGS22, EHL+20]. When working with the Experience API, we can either use a public Registry that is maintained by someone else or create our own and define our own activities, verbs and extensions with their respective definitions and descriptions [HEGS22, EHL+20, KR16]. For more details on the topic of xAPI Registries please refer to the following references: [EHL+20, HEGS22].

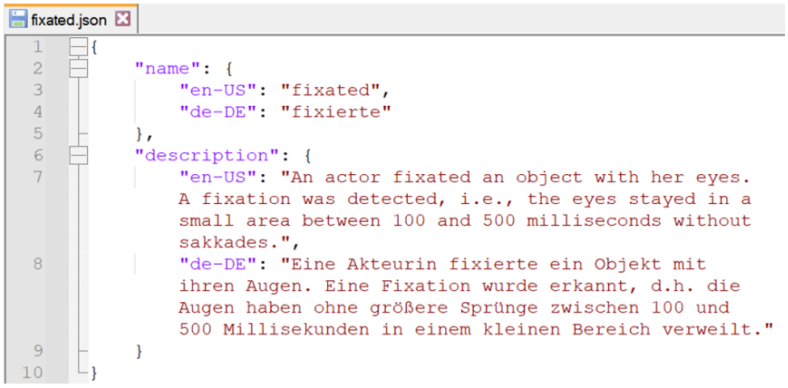

Figure 2.3: Example JSON file: description of the verb "fixated"
Image origin: "xAPI Made Easy: A Learning Analytics Infrastructure for Interdisciplinary Projects" [HEGS22]

[5] `https://xapi.com/registry/?utm_source=google&utm_medium=natural_search`, Accessed on March 1, 2025

2.3 Extended Reality

Extended Reality, or XR for short, is a general term that is used to combine Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR) [ACISIAA22]. These technologies change how we see and interact with the world by mixing reality and digital objects [ACISIAA22]. Instead of looking at a screen, Extended Reality lets users "enter" digital worlds, interact with virtual objects, and experience things that are not necessarily possible or easily doable in real life [ACISIAA22]. For that reason, XR technologies are categorized as fully immersive (completely removing the user from the real world), immersive (combining the real world with virtual elements), or non-immersive (e.g., usual desktop applications) [ACISIAA22].

Virtual Reality is probably the most well-known form of XR. Users typically need a device to enter the virtual application (VR headset) [ACISIAA22]. Users can often move around such virtual worlds and interact with them in different ways. This technology has shown potential in multiple industries [ACISIAA22, Alh16]. For example, medical students can practice surgeries in virtual environments [TTFA24], pilots can train in flight simulators [RG23], engineers can work in virtual labs [AII21, AAW11] etc.

Augmented Reality, on the other hand, keeps the user in the real world but adds virtual elements to it [ACISIAA22]. It does not necessarily require the same hardware that is needed for fully immersive VR environments, but there are also AR glasses (e.g. Microsoft HoloLens). Other types of AR applications work directly on a screen (e.g., *Pokémon Go*) [ALR20].

Lastly, **Mixed Reality** is the middle term. It is used to describe the combination of VR and AR technologies [ACISIAA22].

Extended Reality can be used in different areas for multiple purposes, making certain tasks easier, safer, and generally more engaging [ALR20, SVPG16, BMCC13]. Just to name one example, a major aspect of our lives it could help improve is education [ACISIAA22]. Traditional learning often relies on books or lectures, but these methods don’t always help students fully understand the material [Alh16, ACISIAA22]. Extended Reality can be used to add visual elements to the learning process, making complex topics easier to understand in comparison to text alone [Alh16]. While this adoption of XR technology in education is still in its early stages, some interesting progress is already being made in the field [ACISIAA22].

Extended Reality is certainly promising, but it does have limitations [CAH24, PGG+24]. The most obvious one is the high cost of equipment. However, problems like that could be solved as the technology progresses. Once it becomes more accessible and affordable, we would be able to see its full potential. Today, even though the technology is still developing, its potential is becoming evident [Alh16, ACISIAA22].

2.4 WebAssembly

With the rapid advancements in technology, especially internet technology, we can expect complex applications like games or image and video editing software to also become common on the web [HRS+17, TDO+24]. Such applications, however, require more processing power than the usual website or SPA [HRS+17, TDO+24].

Currently, JavaScript is probably the most used programming language for web development [HRS+17, Hyt23, YTZ+21]. While the language itself and JavaScript engines have been improved quite a bit over time, improvements could be made to bring computationally heavy applications on the web [HRS+17, YTZ+21, TDO+24]. One way to solve that issue is to develop the application in a different language and compile it to a language that can be run on the web [HRS+17]. Writing complex, resource-consuming applications in a language like C or C++ and running them on the web, however, is difficult [HRS+17]. There are many reasons for this, but to name a few we could mention issues like memory management (i.e. one language may be working with manual memory allocation and others may rely on garbage collectors instead), problems with multi-threading or access to the memory and so on [HRS+17, TDO+24, YTZ+21, Hyt23]. Early efforts to solve this problem have developed solutions like *asm.js*[6] [VESN+17, HRS+17] or *Native Client* [YSD+10, HRS+17]. Nevertheless, for various reasons that are beyond the scope of this thesis, these early solutions proved to be insufficient [HRS+17].

Brief introduction to WebAssembly and its advantages

Then, **WebAssembly**[7] (abbreviated as WASM) was introduced. It was developed by a collaboration of engineers from some of the major tech companies which have developed their own browsers (Google, Mozilla, Microsoft etc.) [HRS+17, RB17]. WebAssembly is a low-level, binary instruction format that uses a stack-based virtual machine [HRS+17, YTZ+21, Hyt23]. It is intended to be a compilation target for languages like C, C++, Rust, C# thus allowing the development of complex applications that require higher computing power for the web while also aiming to not be dependent on any language [HRS+17, RB17, TDO+24]. A WASM program is structured as a module that defines functions, variables, memory, and other components [HRS+17, YTZ+21, TDO+24]. These modules can then export functions and other resources to be used by the applications [HRS+17, YTZ+21, MWJR19]. The goal of WebAssembly is not to replace JavaScript, however, but rather to work together with it and enable the development of computationally heavy applications on the web [HRS+17, RB17, TDO+24]. Even when using WebAssembly, a developer would still need to write some JavaScript code (e.g. to import and load the WASM files, to access the DOM Element etc.) [HRS+17, RB17].

In terms of performance, there have been many research studies that have tested WASM's capabilities in comparison to both JavaScript and native languages (like native C) [HRS+17, TDO+24, JPBG19, DMAPS21, YTZ+21, SHH18, Hyt23]. For example, a comparison between WebAssembly and JavaScript can be found in the following references: [YTZ+21, HLP21, DMAPS21]. In these research papers, different tests and measurements in different environments are performed, but what can ultimately be concluded is that, while WebAssembly certainly can bring speed and efficiency to a web application, the performance is also case-dependent [YTZ+21, HRS+17]. For example, this study [YTZ+21] noted that although WASM delivered faster execution times for small input data sets, it actually underperformed when processing large input data sets in comparison to JavaScript. A possible reason for this could be the way WebAssembly manages memory access and memory allocation [YTZ+21, RB17, Hyt23]. Furthermore, the same study also found that WASM seems

[6] `http://asmjs.org/`, Accessed on March 2, 2025
[7] `https://www.w3.org/TR/wasm-core-2/`, Accessed on March 2, 2025

to be using significantly more memory (compared to JavaScript) [YTZ+21]. However, it must be noted that there are efforts to mitigate such issues by, for example, adding garbage collection mechanisms to WASM (please refer to the following websites for more information: `https://github.com/WebAssembly/gc` (Accessed on March 2, 2025) or `https://developer.chrome.com/blog/wasmgc` (Accessed on March 2, 2025)). Nevertheless, it has to also be said that WASM's performance is not easy to measure as it is dependent on multiple outside factors like the chosen compiler or the task itself [YTZ+21, SHH18, Hyt23]. For example, the following research paper [TDO+24] describes positive results when comparing WASM applications with native languages on criteria like execution time, memory usage, CPU load, etc.
While WebAssembly certainly is a promising technology, it also has some disadvantages [WRP+19, MWJR19, HRS+17, YTZ+21, Hyt23]. For example, an issue that seems persistent across different studies is the memory overhead it creates [HRS+17, MWJR19, TDO+24, Hyt23]. Another less obvious problem with WASM is that it can be used with malicious intent as well (e.g. cryptojacking - using the computing power of a user on your website to mine cryptocurrency) [BAA+22, MWJR19, Hyt23].
Despite all that, today, there are web applications that are built with the help of WebAssembly [HRS+17]. For example, the AutoCAD[8] web app is a notable example. A list of more such web applications can be found in the following website: `https://madewithwebassembly.com/`.

2.5 WebXR

In the previous section we reviewed WebAssembly and the efforts behind it to make web applications faster while also increasing complexity by allowing the hosting of computationally heavy programs on the web [HRS+17]. Another category of these complex web applications are ones that incorporate Extended Reality, e.g., VR games, training simulations, learning applications and so on. [IGG+10, MS18]
Early efforts to enable the development of such applications led to the creation of the WebVR API[9] [DUKS18]. The WebVR API enabled functionalities like detecting and communicating with the connected VR devices, rendering 3D scenes, tracking the user's movements and position (to the extent that the VR technology of the time allowed to) etc. [DUKS18]. Furthermore, to enable the incorporation of AR on the web as well, work was also done in the field of WebAR[10] (Web Augmented Reality) [SMY+21]. Nevertheless, both technologies had their problems [MS18]. One such issue was that of standardization [MS18].
Then, in 2018, **WebXR** (Web Extended Reality) was released by a team from the World Wide Web Consortium (W3C) with the idea to mitigate these issues [MS18]. According to the official documentation[11], WebXR is a collection of standards that enable the usage of XR devices and the rendering of 3D scenes in web applications providing a single API for both Virtual Reality and Augmented Reality [MS18]. It is designed to support a wide range of XR devices and is itself supported by most major browsers on the market [MS18]. The core functionalities of the technology are provided by the

[8] `https://www.autodesk.com/products/autocad-web/overview`, Accessed on March 2, 2025
[9] `https://developer.mozilla.org/en-US/docs/Web/API/WebVR_API`, Accessed on March 4, 2025
[10] `https://www.whatiswebar.com/`, Accessed on March 4, 2025.
[11] `https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API`, Accessed on March 5, 2025

WebXR Device API[12]. These include things like the ability to detect and communicate with compatible VR/AR hardware, render the scenes at the appropriate frame rate for the device, positional tracking etc.
Another essential technology that enables the creation of XR applications for the web is *WebGL*[13] (Web Graphics Library). WebGL is a JavaScript API designed for the rendering of 2D/3D graphics in a web browser [YLF+23]. Once a web application creates a WebXR Session, WebGL is used to draw the (3D) elements, to establish communication with the GPU and so on [YLF+23, MS18]. It is essential for any WebXR application as it also is widely supported [MS18].

Brief introduction to different WebXR development frameworks

So far, we established that the WebXR Device API is the connection between the XR devices, the user and the browser [MS18]. Then, WebGL handles the rendering of the virtual elements. When it comes to the development of the actual WebXR applications, however, there are many frameworks that have been developed over the years. Notable examples include frameworks like *Three.js*[14], *A-Frame*[15] and *Babylon.js*[16]. Three.js and Babylon.js share certain similarities in terms of their approach and in the fact that they both rely on JavaScript [Joh21, KN18]. A-Frame on the other hand is written in an HTML-like (declarative) syntax. For the project of this thesis we will keep the focus on *Three.js*, *Babylon.js* and *A-Frame*. Since the topic is to explore different concepts for developing a framework that integrates Learning Analytics into WebXR applications, it is important to understand the similarities and differences between the frameworks. Each framework provides a similar way of structuring the 3D applications with some of the fundamental components being things like scene management, rendering, object creation, animation etc. [D+14, NPNP17, Joh21, KN18]. The basic approach could look like this - we define a scene, place a camera to view it, add the 3D objects to that scene, apply materials and lighting, and use a loop to update the frames.
Creating a **scene** is typically the starting point when developing a virtual application with one of the three frameworks. In Three.js, it could look something like this: `const scene = new THREE.Scene();`. Similarly, in Babylon.js, the scene could be instantiated as follows: `const scene = new Scene(engine);`. A-Frame, on the other hand, defines the scene in a declarative way using HTML (e.g., `<a-scene background="color: white">`) and then the rest of the app is typically written inside that tag. Another essential requirement of a 3D application is the **camera**, which determines how the scene is viewed [D+14, NPNP17]. Then, another important step is the actual creation the 3D objects. In their most basic form, 3D objects have a *geometry* (shape) and *material* (appearance) which is what defines how these objects are viewed by the user [D+14, NPNP17]. While this is an oversimplification, the "interactable" objects that the user can see are typically called *Meshes* as they are usually an instance of a "Mesh" class [D+14, NPNP17]. Three.js requires defining both geometry and material separately before combining them into a mesh, as shown in Listing 2.1.

[12] `https://github.com/immersive-web/webxr/blob/master/explainer.md`, Accessed on March 5, 2025.
[13] `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API`, Accessed on March 5, 2025
[14] `https://threejs.org/`, Accessed on March 6, 2025.
[15] `https://aframe.io/`, Accessed on March 6, 2025.
[16] `https://www.babylonjs.com/`, Accessed on March 6, 2025.

```
const scene = new THREE.Scene();
const geometry = new THREE.BoxGeometry(1, 0.3, 1);
const material = new THREE.MeshStandardMaterial({ color: 0x00aaff });
const button = new THREE.Mesh(geometry, material);
scene.add(button);
```

Listing 2.1: Creating a button mesh in Three.js

Babylon.js, on the other hand, simplifies this by using a *MeshBuilder* (e.g. `const button = BABYLON.MeshBuilder.CreateBox("button", width: 1, height: 0.3, depth: 1, scene);`). Then, for A-Frame, object creation is shown in Listing 2.2.

```
<a-scene>
  <a-box
    position="0 1 -3"
    width="1"
    height="0.3"
    depth="1"
    color="#00aaff"
    class="clickable">
  </a-box>
</a-scene>
```

Listing 2.2: Creating a simple 3D Button in A-Frame

Notice how despite using different approaches to create objects, each framework relies on adding those objects to the scene (see Figure 2.4 for clarity). This similarity between the frameworks is something we leverage in the project presented in this paper.

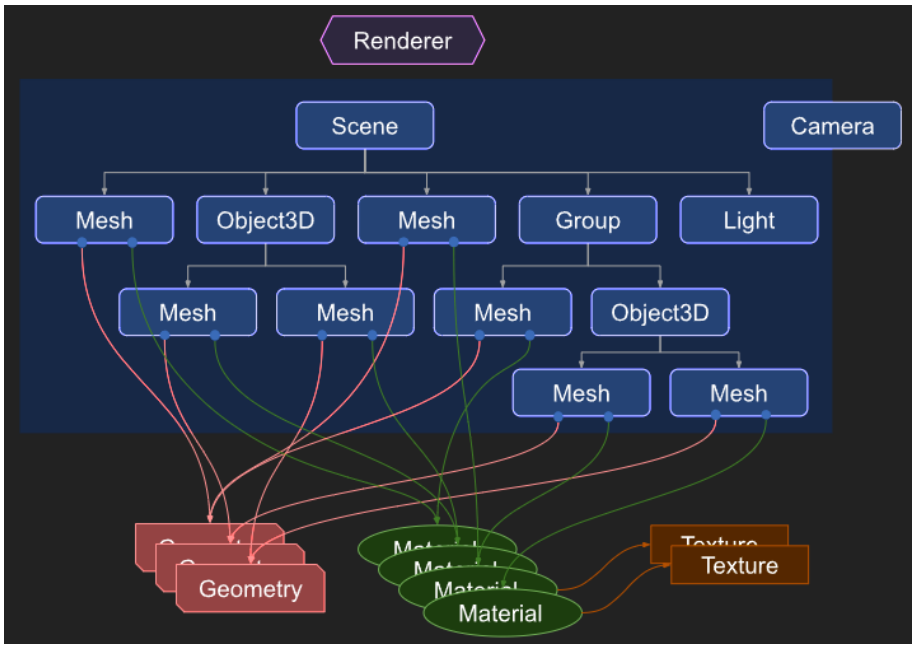

Figure 2.4: Representation of the basic structure of 3D application created with Three.js. Notice how the **Scene** element is at the root of the graph. We will take advantage of that tree-like structure in our version of OmiLAXR.
Image origin: `https://threejs.org/manual/#en/fundamentals`, Accessed on March 5, 2025

Chapter 3 Open and Modular Integration of Learning Analytics in XR (OmiLAXR)

With the increasing potential of Extended Reality applications for learning or training environments (e.g., simulating scenarios that would be hard, expensive or dangerous to test in real life), integrating the technology even more is the logical next step in enhancing education [ACISIAA22, Alh16, GHS24b]. This is not an easy thing to do, however. There are many challenges and problems that would first have to be solved [ACISIAA22, GHS24b, GHS24a]. Besides the complexity of developing XR applications itself, there are also difficulties like the high cost of equipment, keeping up with the advancements in the sector or just plain technical problems that need to be considered [ACISIAA22, GHS24a]. Another less obvious challenge, is how to measure the effectiveness of a given XR learning application once all the technical issues have been solved [LC25]. This is exactly what the OmiLAXR framework is designed for [GHS24b, GHS25].

The *OmiLAXR* framework (short for **Open and Modular Integration of Learning Analytics in XR)** is designed as a solution to the aforementioned challenge of integrating Learning Analytics in XR [GHS24a, GHS24b]. It attempts to standardize and automate the collection of Learning Analytics data in a virtual environment by providing a simple "*Plug and Play*" concept [GHS24a, GHS24b]. Simply stating that it only gathers information about how a user progresses through an application however is an oversimplification [GHS24b, GHS25]. The goal of OmiLAXR is to make the tracking of user interactions and experiences in an XR application seamless and easy to use while also providing complex analytics about it [GHS24a, GHS24b]. Keeping track of such activities in a virtual application is not a simple task, however. [GHS24a, GHS24b]. Furthermore, the goal is to not only track simple analytics like completion time or scores but also more sophisticated information like positional tracking, gestures and more [GHS24a, GHS24b]. Therefor, an important part of the OmiLAXR framework was to make the collected data as informative as possible, thus allowing for complex data analysis while also using a standard that is compliant with the FAIR principle (findability, accessibility, interoperability, and re-usability) [GHS24a, GHS24b, SBG$^+$24]. For that reason, OmiLAXR relies on the *Experience API* (xAPI) as its standard for data collection [GHS24a, GHS24b, HEGS22, GHS23]. Its way of structuring information - by defining an Actor (e.g., the learner or user), a Verb (e.g., grabbing an object in the virtual environment), and an Object (the item being

interacted with) - has proven to be an effective solution [GHS24a, GHS24b, HEGS22]. The detailed technical aspects of OmiLAXR are beyond the scope of this work, but a general explanation of the framework's design will be provided. It must also be noted that there are 2 versions of the OmiLAXR framework that differ in their core designs. In this chapter, both versions will be presented.

3.1 OmiLAXR version 1

A review of the first version of the project is provided in the following references: [GHS24a, GHS24b]. Ultimately, this version relies on a few different components that work together and communicate with each other to achieve the desired functionality [GHS24b]. A general representation is provided in Figure 3.1. Essentially, these components can be viewed as separate systems that map the user's (or system's) actions and experiences to structured xAPI statements and store them in a *Learning Record Store* [GHS24a, GHS24b].

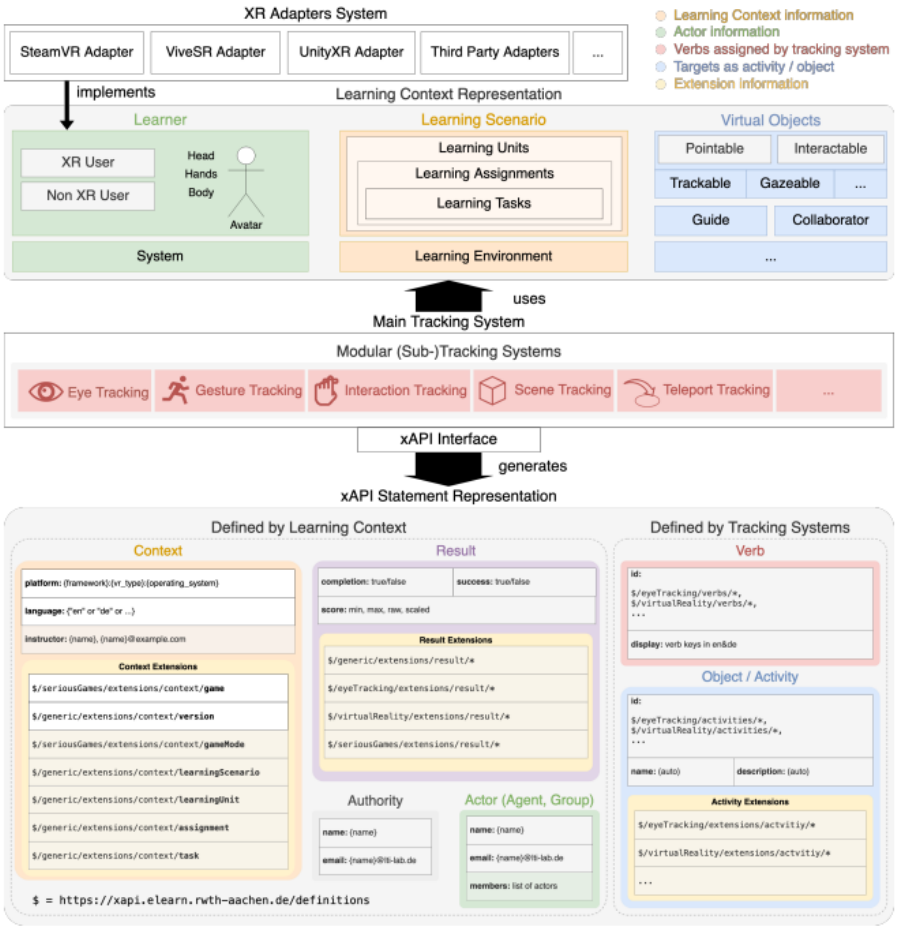

Figure 3.1: An illustation showing the concept of the first version of the OmiLAXR framework. Image origin: "Towards using the xAPI specification for Learning Analytics in Virtual Reality" [GHS24b]

1. **XR Adapters System**
   First, there is the XR Adapters System. As previously stated, an important quality of OmiLAXR is its *"Plug and Play"* concept [GHS24b]. The role of the adapters is related to that principle [GHS24b]. The XR Adapters System acts as a "*middleman*" between the different frameworks and the rest of OmiLAXR's components by abstracting challenges like device-specific input systems, for example [GHS24b]. With its help, developers do not need to worry about the framework or hardware their applications will be used with as they can just use a new adapter instead of having to rewrite the code [GHS24b]. That adapter then enables the communication between the tracking logic of OmiLAXR, the Unity application and the respective VR hardware [GHS24b].

2. **Learning Context Representation**
   In the component for Learning Context Representation is where the important questions are answered [GHS24b]. These are questions like the *"who?"* (who is the actor of the tracked action), the *"what?"* (what action exactly was just tracked), the *"what on?"* (what object from the virtual environment was the tracked action performed on exactly) and also further details about the interaction (context) [GHS24b, CDST12]. As showcased in Figure 3.1, here OmiLAXR defines entities such as the learner, learning scenario, virtual objects, and task hierarchy (e.g., units, assignments, etc.), while also allowing developers to mark relevant elements as *trackable* or *gazeable* [GHS24b]. This functionality serves the purpose of linking the resulting xAPI statement to the respective context so that detailed analysis on the data can be performed [GHS24b, GHS24a].

3. **Main Tracking System**
   The Main Tracking System is responsible for the detection, capturing and organizing of interactions in the virtual environment to xAPI statements [GHS24b, GHS24a]. As showcased in Figure 3.1, it provides a variety of different "subsystems" for different actions and experiences that are possible inside an XR application (e.g. gesture tracking, interaction tracking, teleporting, etc.) [GHS24b]. Furthermore, the main tracking system is designed to work with the previous components [GHS24b]. Once an action (e.g., grabbing an object) is detected, it can pull details about the learning context using them to make the statement more informative [GHS24b]. That information is then combined with the details about the user, the verb, and the object and an xAPI statement is created [GHS24b].

4. **xAPI Statement Representation**
   Previously, in the "Background" chapter, we explained the importance of standardization, particularly in the field of xAPI [HEGS22, GHS24b]. For that reason, the creators of OmiLAXR also maintain their own xAPI registry[1].
   As displayed in Figure 3.1, the **xAPI Statement Representation** part of the figure showcases in more detail how the final xAPI statements are created.

[1] `https://xapi.elearn.rwth-aachen.de/` Accessed on March 11, 2025.

5. **Evaluating the framework**
   The team behind OmiLAXR also performed a small study to evaluate the framework based on the following metrics: productivity, workflow, usability, and functionality and reliability [GHS24a, GHS24b]. Here, we will only present a brief summary of the results but a more detailed review of the evaluation is provided in the following references: [GHS24a, GHS23, GHS24b]. Ultimately, seven computer science master's students were asked to use OmiLAXR in a VR application and report on their experiences of using the framework [GHS24a]. To track the way the students used OmiLAXR, the creators monitored their activities and also offered a crash course to ensure foundational knowledge of Unity and the chosen VR project for the test [GHS24a]. At the beginning, the students struggled with the initial setup for OmiLAXR, but once they figured it out, the results were positive [GHS24a]. In accordance with the evaluation criteria, the creators concluded that:

   a) **Productivity** - The framework worked quite well. Over 250,000 xAPI statements have been collected, reporting that the prebuilt tracking mechanisms worked as intended and the creation of new functionalities was simple and efficient [GHS24a, GHS24b, GHS23].

   b) **Workflow** - The paper ([GHS24a]) mentions that a suggested workflow had been provided in the official documentation[2] but some students decided to create their own workflow by adding new features to the framework instead [GHS24a].

   c) **Usability** - As mentioned earlier, the students did struggle with the initial setup of OmiLAXR. The paper ([GHS24a]) explains that the participants in the study asked for more examples and prebuilt components to aid the understanding of how the framework works. The authors also noted that the complexity of the project and the incomplete documentation (at the time of the study) was an issue [GHS24a, GHS23, GHS24b].

   d) **Functionality and Reliability** - As far as this criterion goes, the researchers noted that validating the data was not efficient enough and that they plan on developing a monitoring tool to mitigate the issue [GHS24a, GHS23, GHS24b].

3.2 OmiLAXR version 2

The second version of OmiLAXR redefines the core concept of the original design. This new implementation relies on a pipeline-like structure (see Figure 3.2) [GHS25, Gör25a, Gör25b]. The pipeline consists of six components - **Listeners**, **Filters**, **Tracking Behaviours**, **Composers**, **Hooks** and **Endpoints** [GHS25, Gör25a]. Now, as showcased in Figure 3.2, the listeners and the filters answer the question *"what are we tracking exactly?"*, the tracking behaviours - *"when is an action to be tracked?"*, the composers and the hooks - *"how do we structure the gathered data?"* and, lastly, the endpoints - *"where do we store the collected information?"* [GHS25, Gör25a, CDST12]. Furthermore, the six components are grouped

[2] `https://gitlab.com/learntech-rwth/omilaxr-ecosystem/omilaxr-documentation/-/releases/Nuts-and-Bolts-2024-Publication` Accessed on March11, 2025.

into two bigger categories - the **Actor Pipeline(s)** and the **Data Provider(s)** [GHS25, Gör25a]. The *Actor Pipelines* are responsible for the tracking of interactions or experiences and the *Data Providers* handle the collected data and structure it in the respective format (e.g., xAPI statements) thus separating the fields of data tracking and data processing [GHS25, Gör25a]. This approach makes it easier for developers to add or remove components based on their needs (the *"Plug and Play"* concept) and also allows for the creation of custom components that follow the respective base structure [GHS25, Gör25a]. A detailed review of each component is provided on the official website[3] ([Gör25a]) and in the following research paper [GHS25]. Nevertheless, a brief explanation will also be provided below:

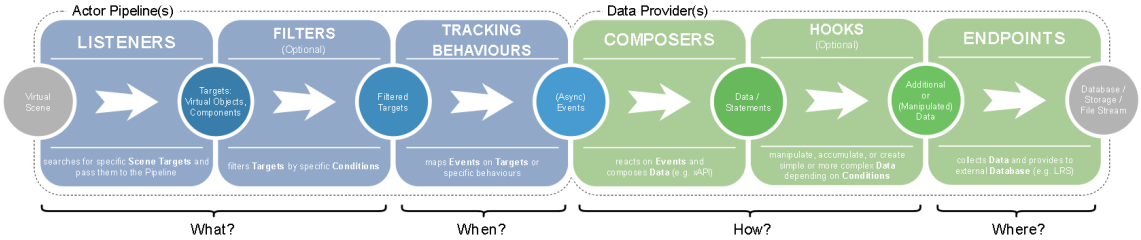

Figure 3.2: The OmiLAXR Pipeline design showing the six components and the split into Actor Pipelines and Data Providers.
Image origin: https://omilaxr.dev/pipeline/ [GHS25, Gör25a]

1. **Listeners** - In this version, OmiLAXR uses listeners to "scan" the virtual environment for specific "target" objects [GHS25, Gör25a]. Developers can use prebuilt listeners (e.g. "ColliderListener" for detecting GameObjects that have the Collider component as showcased on the website ([Gör25a]) or define their own [GHS25].

2. **Filters** - Filters receive the "scanned" objects from the listeners and provide the ability to filter out certain elements from the passed set [GHS25, Gör25a]. Developers can again use prebuilt filters or define their own. Application of multiple filters on the same set of objects is also possible, but it must be done with caution as the order would be important in this scenario [GHS25, Gör25a].

3. **Tracking Behaviours** - As the name suggests, OmiLAXR uses Tracking Behaviors to detect and react on interactions with the objects that have passed the filters [GHS25, Gör25a]. Once an action is detected by a Tracking Behaviour, it emits public events which can then be used by the next component of the pipeline [GHS25, Gör25a].

4. **Composers** - This stage of the pipeline is where the detected interactions are *composed* into a structured data format (xAPI statements or some other format) based on the events from the Tracking Behaviors [Gör25a]. Here we are also introduced to another useful functionality of the framework - making xAPI integration simple and beginner-friendly [Gör25a, GHS25]. In the OmiLAXR ecosystem, developers can use the "actor.Does(verb).Activity(activity).WithExtension(activityExtension)" syntax and

[3] `https://omilaxr.dev/pipeline/` Accessed on March 11, 2025.

not have to worry about any in-depth specifics of the TinCan (Experience API) library [Gör25a].

5. **Hooks** - OmiLAXR uses hooks to allow developers to modify the data as structured by the composers (in this case the xAPI statements) before finalizing it [GHS25, Gör25a]. For example, hooks can be used to add context information to the statements [GHS25, Gör25a].

6. **Endpoints** - The main responsibility of the OmiLAXR endpoints is to ensure the storage of the collected data [GHS25, Gör25a]. This could be, for example, the local hard drive, a database or (in our case) a Learning Record Store [GHS25, Gör25a].

OmiLAXR was designed to be modular and extensible - for each component, developers can either use prebuilt ones or define their own implementations [GHS25, Gör25a]. Furthermore, it also allows to have multiple Actor Pipeline or Data Provider instances [GHS25, Gör25a]. For example, we could use one Actor Pipeline to track user interactions (e.g., grabbing an object, teleporting etc.) and another to track system related actions (e.g., game start, changes in the virtual environment etc.), and we could also have a Data Provider instance that creates xAPI statements and another that structures the data in a custom format of our choice [GHS25, Gör25a]. A formal comparison between the two versions of OmiLAXR is beyond the scope of this thesis, however.

Chapter 4 Realization of the OmiLAXR Framework for WebXR Environments

After presenting the necessary pre-knowledge and reviewing the design and technical aspects of the original OmiLAXR, we are ready to present the core topic of this paper and review the **research question** we aim to explore - "*How can the OmiLAXR framework be realized in WebXR environments?*". Essentially, this work explores different possibilities of how the functionality of OmiLAXR can be adapted to WebXR applications. That way, we can show the framework's independence from the development platform (e.g., applications developed with Unity, WebXR etc.) and also identify challenges in the process of realizing it for WebXR applications.
In our attempts to achieve these goals, two projects were developed. The first idea was to use C# (compiled into WebAssembly) for the pipeline structure and JavaScript for the communication with the browser and the WebXR Device API. Moreover, as the WebXR development frameworks we are focusing on (e.g., Three.js, Babylon.js, A-Frame, etc.) are based on JavaScript, the idea was to ultimately bundle the project into a JavaScript library (using a tool called **Rollup.js**) which would allow developers to easily import and use the project to integrate Learning Analytics into their XR applications.
The second approach we realized was almost entirely created using TypeScript instead. Ultimately, this is also the design that we decided best fits the goals of the project and the one we chose to continue developing. As far as the technical concepts go, there are no major changes besides the difference between the programming languages. The TypeScript version of the framework again adopts the pipeline architecture and is also bundled into an importable library that enables developers to easily setup and create instances of the pipeline and also configure it according to their needs (e.g., add or remove components as needed).
In this chapter both approaches will be presented and briefly reviewed, starting the first version.

4.1 OmiLAXR for WebXR - approach 1

In this section, we present and review the first concept we developed to adapt the OmiLAXR framework for WebXR environments using C#, WebAssembly and Javascript.

When designing a development plan, one of the goals for the project was to reuse the core interfaces (e.g. the pipeline interfaces) from the original OmiLAXR framework (created with C#) so that developers can easily switch from one version to the other as they would already be familiar with the main technicalities. An interesting challenge in the beginning, however, was the question of how to use C# on the web. The solution we decided best fits our goals was WebAssembly. For one, we hoped to leverage the speed of WASM on the web and to also make use of the multithreading capabilities of C# and the .NET ecosystem. Different ways of compiling C# applications into WASM have already been developed (e.g. Blazor WebAssembly). For this project, a tool called *Bootsharp*[1] was chosen. In short, it enables the integration of C# code into JavaScript and TypeScript web applications by bundling and compiling a given .NET application into an importable WebAssembly file. The tool works by allowing developers to "mark" certain methods (or events) that can then be called from within the JavaScript program. The way this "marking" works is via an attribute (e.g., [JSInvokable]). After the compilation to WASM, the marked methods (or events) will be exported by the resulting WASM file (usually called "index.mjs"). Another relevant question is also why we did not opt for Blazor WebAssembly[2], which is an established framework, backed by Microsoft. It would offer advantages like long-term support, reliability, a developed ecosystem etc. After some initial testing, we decided that Blazor WASM does not really fit into the goals of this project. With that being said, a simplified review of the project will be presented below:
As in the OmiLAXR concept, we adopt the pipeline approach. Initially, we take *Three.js* as an example to build the framework.
Before we begin with the code review and explain how the project works, we will explain how it is created as it involves different technologies and tools. Initially, a .NET console application is created (via `dotnet new console -n OmiLAXR`). This creates the basic structure of the application. The main files we are dealing with are the **Program.cs** file, which serves as the entry point of the .NET application and contains the *Main* method and also the *OmiLAXR.csproj* file that defines the project dependencies, versions, target framework, etc. Then, Bootsharp is installed as a NuGet package[3] (via `dotnet add package Bootsharp -version 0.3.3`). Version 0.3.3 was the latest at the time of creating the project, but as of the time of writing this paper (April 2025) a newer version (0.6.0) has been made available.

4.1.1 Code review

Before discussing the results of this approach, we will first review parts of the code that implement the pipeline logic.

The Listeners

The **Listener** class here is responsible for processing raw data and converting it into a structured format - an instance of the *ObjectOfInterest* class is used to standardize the communication within the pipeline. The *Listen* method takes a list of lists of strings where each inner list holds information about an object from the virtual

[1] `https://sharp.elringus.com/guide/`, Accessed on March 15, 2025
[2] `https://blazorise.com/blog/what-is-blazor-wasm`, Accessed on March 15, 2025
[3] `https://www.nuget.org/packages/Bootsharp`, Accessed on March 15, 2025

environment (e.g. ObjectID or Tag). The method iterates over the *objectsData* list, creating new instances of type ObjectOfInterest for each "scanned" virtual element from the environment and returns a list of these instances. An example of this functionality is displayed in Listing 4.1.

```
1
2  public class DefaultListener : BaseListener
3  {
4      public override List<ObjectOfInterest> Listen(List<List<string>>
           objectsData)
5      {
6          var objects = new List<ObjectOfInterest>();
7          foreach (var objData in objectsData)
8          {
9              objects.Add(new ObjectOfInterest(objData[0], objData[1], objData
                   [2]));
10         }
11         return objects;
12     }
13 }
```

Listing 4.1: DefaultListener implementation in approach 1

The Filters

The **Filter** class receives that list from the Listener and refines it by excluding each object not "marked" for tracking in its *Apply* method. Since developers may not be interested in tracking every single object in the environment, they need a way to tell OmiLAXR which objects are to be monitored. Here, the custom *tag* property of each object is used for that. Developers can specify a tag of their choice and pass it to OmiLAXR, as we will later see. The filtering logic (based on the tag) is shown in Listing 4.2.

```
1  public class TagFilter : BaseFilter
2  {
3      public override List<ObjectOfInterest> Apply(List<ObjectOfInterest> objects
           , string tagToTrack)
4      {
5          return objects.FindAll(obj => obj.Tag == tagToTrack);
6      }
7  }
```

Listing 4.2: TagFilter class that filters objects based on a specific tag

The TrackingBehaviors

The **TrackingBehavior** class is the key component of the Actor Pipeline part of the framework. Initially, we focused on tracking mouse clicks. The way this class functions is it takes the filtered list of ObjectsOfInterest and creates structured metadata about the detected interactions of type *InteractionMetadata*. An example implementation of a Tracking Behavior is shown in Listing 4.3.

```
public class TrackingBehavior : BaseTrackingBehavior
{
    public override List<InteractionMetadata> GenerateMetadata(List<
        ObjectOfInterest> objectsToTrack)
    {
        var metadataList = new List<InteractionMetadata>();
        foreach (var obj in objectsToTrack)
        {
            metadataList.Add(new InteractionMetadata(obj.ObjectID, obj.Name, 0)
                );
        }
        return metadataList;
    }
}
```

Listing 4.3: Basic TrackingBehavior implementation in approach 1

The Composers

The **Composer** class is responsible for constructing structured statements about the detected interactions. For this, it uses the list of *InteractionMetadata* objects it receives from the *TrackingBehaviors*. This implementation of the Composer, called *BaseComposer* (shown in Listing 4.4), was designed for testing purposes, i.e. the generated statements do not follow the xAPI standard but are designed to be human-readable instead. Nevertheless, the project can easily be extended to also offer a Composer that implements the TinCan.NET[4] library.

```
public class Composer : BaseComposer
{
    public override string ComposeInteraction(string userID, string objectName,
         string action)
    {
        // Example statement: "TestUser clicked on StartEngineButton"
        return $"{userID} {action} on {objectName}";
    }
}
```

Listing 4.4: Composer class that formats interaction data into a readable statement

The Hooks

The **Hook** class is responsible for enriching the resulting statement with additional information (in this case timestamp information at which the interaction occurred). Should developers wish to include that information to their statements, they need to set the *includeTimestamp* variable to true. The *TimestampHook* class is shown in Listing 4.5.

```
public class TimestampHook : BaseHook
{
```

[4] `https://github.com/RusticiSoftware/TinCan.NET`, Accessed on March 15, 2025

```
    public override string AddInfo(string statement, string timestamp, bool
        includeTimestamp)
    {
        return includeTimestamp ? $"{statement} (timestamp: {timestamp})" :
            statement;
    }
}
```

Listing 4.5: TimestampHook class that adds a timestamp to the statements

The Endpoints

The **Endpoint** class is responsible for the storage of the final statements. Ultimately, the statements can be stored at the local storage, at a database (LRS) or printed out in the console as shown in the ConsoleEndpoint example (Listing 4.6).

```
public class ConsoleEndpoint : BaseEndpoint
{
    public override void Send(string finalStatement)
    {
        Console.WriteLine(finalStatement);
    }
}
```

Listing 4.6: ConsoleEndpoint class that logs the final statement to the console

Then, we also have a class to represent the *ActorPipeline* and one for the *DataProviders*.

The Actor Pipeline

The **ActorPipeline** class, displayed in Listing 4.7, acts as a "manager" for the first three components in the pipeline. Ultimately, it serves the purpose of holding the instances of the three components running them in the respective order. The key functionality here is the *StartActorPipeline* method which takes the list of lists and runs it through the components. This will be one of the methods we exported via the WebAssembly file.

```
public class ActorPipeline
{
    public List<InteractionMetadata> StartActorPipeline(List<List<string>>
        objectsData, string tagToTrack)
    {
        var objects = _listener.Listen(objectsData);
        var filteredObjects = _filter.Apply(objects, tagToTrack);
        return _trackingBehavior.GenerateMetadata(filteredObjects);
    }
}
```

Listing 4.7: ActorPipeline class coordinating the listener, filter, and tracking behavior

The Data Providers

Then, the **DataProviders** class, shown in Listing 4.8 "manages" the last three components. Again it has a *StartDataProviders* method which takes the necessary input data for each component and runs them in the correct order. That method will also be exported by the WebAssembly file.

```
public class DataProviders
{
    public void StartDataProviders(string userID, string objectName, string
        action, string timestamp, bool includeTimestamp)
    {
        var statement = _composer.ComposeInteraction(userID, objectName, action
            );
        var finalStatement = _timestampHook.AddInfo(statement, timestamp,
            includeTimestamp);
        _consoleEndpoint.Send(finalStatement);
    }
}
```

Listing 4.8: DataProviders class coordinating the composer, hook, and endpoint components

The Program class

Lastly, we have the **Program** class. It acts as the entry point for the project and a bridge between the C# and the JavaScript parts of the program. This is where we expose the necessary methods to Bootsharp (via the **[JSInvokable]** attributes as showcased in Listing 4.9) to keep everything in one place, making it easier to track. We also have the *Main* method here, since we experienced compilation errors without it.

```
public class Program
{
    [JSInvokable]
    public static List<InteractionMetadata> StartActorPipeline(List<List<string
        >> objectsData, string tagToTrack)
    {
        var actorPipeline = new ActorPipeline();
        return actorPipeline.StartActorPipeline(objectsData, tagToTrack);
    }

    [JSInvokable]
    public static void StartDataProviders(string userID, string objectID,
        string action, string timestamp, bool includeTimestamp)
    {
        var dataProviders = new DataProviders();
        dataProviders.StartDataProviders(userID, objectID, action, timestamp,
            includeTimestamp);
    }
}
```

Listing 4.9: Program class exposing the start methods for the Actor Pipeline and Data Providers

Now, once the C# code is ready, we compile it into WebAssembly with a simple .NET CLI command - `dotnet publish -c Release -r browser-wasm`. It tells the .NET SDK to build and package the application and compile it to the specified output format with the .NET runtime ("-r browser-wasm"). Then, a file called "**index.mjs**" is created. That file contains our pipeline logic and exports the necessary methods.

Connecting the Pipeline and the WebXR Device API

One of the goals of the OmiLAXR for WebXR project was to make it simple to use. For that reason, the idea was that a developer would only have to import our library and call a single **startOmilaxr** function to initialize a pipeline and start the tracking process. Hence why, the JavaScript code is designed accordingly. The structure of the startOmilaxr function is shown in Listing 4.10.

```
export async function startOmilaxr(THREE, sceneObj, camera, userID,
    includeTimestamp) {
    // 1) Initialize Bootsharp
    await bootsharp.boot();
    // 2) Gather the virtual objects from the scene
    const objectsData = [];
    sceneObj.traverse((object) => {
        if (object.isMesh) {
            objectsData.push([
                object.uuid,
                object.name || "Unnamed",
                object.userData.tag || "None",
            ]);
        }
    });
    console.log("Scanned objects:", objectsData);
    // 3) Use the ActorPipeline to filter and create metadata
    const metadata = await Program.startActorPipeline(objectsData, "object of
        interest");
    // 4) Attach event listeners (click event in this case)
    attachClickListeners(THREE, sceneObj, camera, metadata, userID,
        includeTimestamp);
}
```

Listing 4.10: startOmilaxr function - initializes the scene and attaches event listeners

Initially we load and start the compiled WebAssembly module. We then continue by using the **scene object** to create an array of the objects in the environment with the necessary data for the pipeline to be able to process them (e.g. object ID and tag). Afterwards we call the exposed **Program.startActorPipeline** method and pass to it the array of objects and the tag to be used by the filters. Lastly, we use the **attachClickListeners** function to create event listeners for the actions we are tracking. The way this works is shown in Listing 4.11.

```
function attachClickListeners(THREE, sceneObj, camera, metadata, userID,
    includeTimestamp) {
    // Build a map from ObjectID -> InteractionMetadata
    const objectMap = new Map(metadata.map((meta) => [meta.ObjectID, meta]));

```

```
    // Create raycast to detect where a mouse click happened

    // Global 'click' event to detect mouse clicks
    window.addEventListener("click", (event) => {
        // 1) Convert mouse screen coords to normalized device coords...
        // 2) Set the raycast from the camera pov
        raycaster.setFromCamera(pointer, camera);
        const intersects = raycaster.intersectObjects(sceneObj.children, true);
        if (intersects.length > 0) {
            // 3) If the first intersected object matches our map, log a
                statement
            const clickedObject = intersects[0].object;
            const meta = objectMap.get(clickedObject.ObjectID);
            if (meta) {
                // 4) Call the DataProviders
                Program.startDataProviders(
                    userID,
                    meta.objectName,  // e.g. object name
                    "clicked",
                    new Date().toISOString(),
                    includeTimestamp
                );
            }
        }
    });
}
```

Listing 4.11: attachClickListeners function for handling click-based interaction tracking

Essentially, this function attaches a global click event and detecting the mouse position at the time of the click. It then checks if the click *intersects* with an object from the metadata list which contains the objects that have been passed through the Actor Pipeline. If the code detects that a click was performed on an object we are tracking (i.e. marked with the "object of interest" tag), it calls the exported **Program.startDataProviders** and a statement about the interaction is logged in the console.
Finally, the project is bundled into a JavaScript library with **Rollup.js**. To showcase how it can be used, a basic test project was created. It contains two cubes representing a *Start Button* and a *Stop Button*. There are other elements as well (e.g. the floor element) which are removed from the list of objects to track by the filters as they are not "marked" with the necessary *tag*. To initiate the framework, developers import the bundled library and call the start method (e.g., `startOmilaxr(THREE, scene, camera, userID, includeTimestamp);`). Then, mouse clicks are automatically being tracked (as long as the click is performed on an *object of interest*). The test application and the resulting statement are shown in Figure 4.1 and Figure 4.2. Notice how, after the initial scan, 5 objects have been detected (in the array under "Scanned objects") while afterwards there are only two left in the **post-filter array** (under "Filtered Metadata").

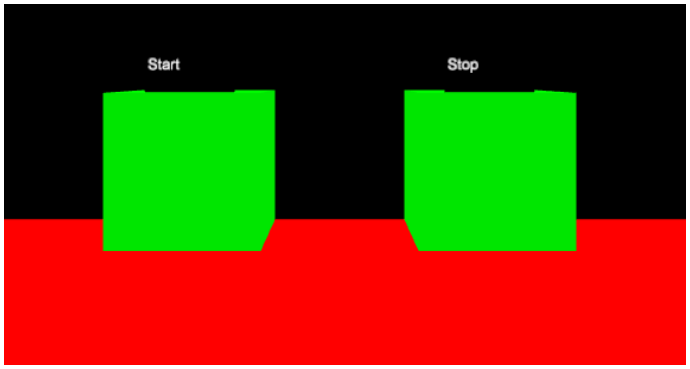

Figure 4.1: Basic Three.js test application with two cubes serving as Buttons. The OmiLAXR library is imported in the application. The image was generated by the author.

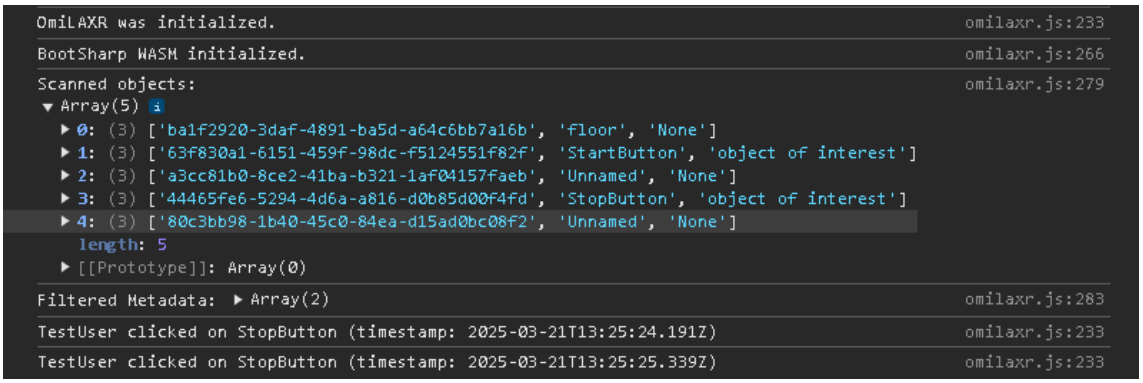

Figure 4.2: Screenshot of the generated xAPI statements in the console after having clicked on each of the buttons with the mouse. The image was generated by the author.

4.1.2 Reviewing the approach

As explained earlier, the main idea behind using this approach was to share the core interfaces (created with C#) of OmiLAXR between the different implementations (e.g., for Unity, for WebXR etc.). That way we would not only continue with a similar architecture but we would also make it easy for developers to switch between different versions of the framework as they would be familiar with the functionality. This version of the project was discontinued, however, due to it not being well-suited to the goals of the framework. The core idea of OmiLAXR is to provide a developer-friendly solution for integrating Learning Analytics while also being easily extensible. In practice, however, this approach proved to be significantly more problematic than anticipated as we faced multiple problems during implementation.
For instance, before we decided to use Bootsharp, we created a version of the framework with Blazor WebAssembly. This, however, led to technical issues like the appearance of web pages, predefined by Blazor, blocking or crashing the Three.js applications used for testing. Afterwards, once we switched to the use of Bootsharp, we managed to mitigate any problems related to the compilation to WASM or the integration of the framework into WebXR applications. Nonetheless, new practical challenges emerged. The main cause for it was the split between C# and JavaScript. For example, adding any custom functionality, such as new TrackingBehaviors or Endpoints required developers to directly modify the original source code of the framework and

recompile it to WebAssembly which was limiting developers' freedom instead of the opposite. Furthermore, although the application was technically split into two parts (C# for the pipeline and JavaScript for the browser communication), in practice, much of the actual functionality (like interacting with the DOM, attaching event listeners, or accessing the WebXR Device API) had to be handled in JavaScript. This led to issues like an unclear separation of concerns, thereby making the framework less developer-friendly, or inefficient data flow between the two parts of the framework. For example, to pass scene data from a Three.js application to the C# pipeline, we had to extract relevant information (e.g., object ID, name, and tag) from every mesh in the scene, transform it into a nested JavaScript array, and pass it to the WebAssembly module. This data was then processed again in C#, restructured into model classes like ObjectOfInterest, and only then used for filtering and tracking. This process introduced unnecessary overhead. Moreover, any errors in the pipeline, whether type mismatches, index errors, or invalid object data, were also difficult to debug.
Considering all the limitations, an important question is why we did not start with the second approach from the beginning. In retrospect, the decision to use C# and WebAssembly was not necessarily wrong. Performance could have been gained from the speed offered by WASM and the multithreading capabilities of C# and the .NET ecosystem. These features could have been leveraged, for example, to handle performance-heavy tasks like the generation and asynchronous sending of xAPI statements to a database, for example. However, this approach proved to not be well-suited for the core values of the OmiLAXR framework (e.g., simplicity, developer-friendliness, flexibility etc.). For that reason, we chose to discontinue this version and pursue a different approach instead.

4.2 OmiLAXR for WebXR - approach 2

This approach was developed using TypeScript. We are again following the pipeline architecture and rely on the Experience API as our data format. Furthermore, the development of this version was also focused on better adapting the "Plug and Play" principle of the original OmiLAXR, i.e. keeping in mind the need to let developers easily add and remove components (or create custom ones) based on their needs and requirements. Lastly, once finished, the code is again bundled into a JavaScript library so that it can be easily imported in various WebXR projects.
In this chapter, the framework will be presented and its functionality explained in more technical terms. We will first provide a brief review of the project structure, however, to ensure clarity. A quick disclaimer before we move on: the project was initially created with a strong focus on *Three.js*, but we will explore how it can be extended to other WebXR development frameworks like *Babylon.js* or *A-Frame* in the latter chapters as well. As showcased in Figure 3.2, the framework tracks interactions in six steps. There is an abstract base class for each component which plays a key role for when developers wish to implement their own custom logic and integrate it into the pipeline. Then, there are two classes that act as *"managers"* for the components, those being the *ActorPipeline* and the *DataProviders* classes (see Figure 3.2). There are "helper" classes for the creation of the xAPI statements and finally there is the *Pipeline* class. The Pipeline class orchestrates the entire functionality and also provides the entry point for developers using the framework by exporting three

interfaces, namely the *ActorPipelineConfig*, *DataProvidersConfig* and *PipelineConfig* interfaces. Developers who import the bundled library essentially create a *Pipeline object*, use the interfaces to configure it (tell OmiLAXR which components exactly they wish to use) and simply call the *pipeline.start* method. Now, we can proceed with the code review below:

4.2.1 Code Review

Once again, before we discuss the results from this approach, we will first provide a review of the code to showcase the way OmiLAXR is realized in WebXR environments using TypeScript.

The Listener class

First, we have the **Listener** class. A prebuilt example implementation we offer, called the **DefaultListener**, is showcased in Listing 4.12. It is responsible for scanning the *scene object*, extracting all the elements in it and structuring them into an array where each entry is an object containing metadata about the "scanned" 3D elements (e.g. ID or tag).

```
export class DefaultListener extends Listener {
  public scan(scene: any): any[] {
    const collectedObjects: any[] = [];

    if (scene && Array.isArray(scene.children)) {
      scene.children.forEach((child: any) => {
        collectedObjects.push({
          id: child.userData?.id || child.id,
          name: child.name || "",
          tag: child.userData?.tag || "",
          reference: child
        });
      });
    }

    return collectedObjects;
  }
}
```

Listing 4.12: DefaultListener implementation for the second approach - this time it does the scanning itself in comparison to approach 1

The Filter class

Next, we have the **Filter** classes. The filters receive the array of "scanned" objects from the listeners and apply predefined filtering logic to it, thus excluding objects that we are not interested in tracking, for example. Listing 4.13, showcases two of our filters - **TagFilter** and **MeshesOnlyFilter**. The TagFilter class again gives developers the option to "mark" (via their *tag* property) certain objects in the virtual environment, thereby configuring the pipeline to only process those objects. On the

other hand, the MeshesOnlyFilter removes all non-Mesh objects from the array (e.g. cameras or lighting related elements) and only leaves the actual 3D objects in it.

```
export class TagFilter extends Filter {
  private tag: string;

  constructor(tag: string) {
    super();
    this.tag = tag;
  }

  public apply(objects: any[]): any[] {
    return objects.filter(obj => obj.tag === this.tag);
  }
}

export class MeshesOnlyFilter extends Filter {
  public apply(objects: any[]): any[] {
    return objects.filter(obj => obj.isMesh === true);
  }
}
```

Listing 4.13: TagFilter and MeshesOnlyFilter implementations

The TrackingBehavior class

Then, we have the **TrackingBehavior** classes. As the name suggests, these classes are responsible for the actual tracking of interactions in the virtual environment. They define how, when, and what kind of interaction will be monitored. Currently, the framework provides prebuilt implementations monitoring mouse clicks and VR controller buttons via the **MouseClickTrackingBehavior** and the **ControllerTrackingBehavior** classes. Furthermore, here (in the base TrackingBehavior class) we define two internal interfaces that play a crucial role in the way OmiLAXR's components communicate with each other. These interfaces are called **InteractionData** and **ExtraData**. The Tracking Behaviors will use instances of these interfaces to pass information to *Data Providers* (see Figure 3.2). The main method of these classes that initiates everything is called *startTracking*. Once an interaction is detected, the respective TrackingBehavior class creates objects based on the two interfaces and stores the necessary information about the interaction in them. For instance, the InteractionData interfaces stores object related metadata (e.g. objectId or the type of interaction that was just detected) while the ExtraData interfaces are used to store additional information like timestamp or, in the case of mouse clicks, the coordinates of mouse at the time of the click. Another important aspect of these classes is their *callback mechanism*. All Tracking Behaviors receive a function we internally call "*onInteraction*" along with their usual arguments (e.g. the array of objects that have passed the filters). They use that function to pass information to the next component (by passing the two *InteractionData* and *ExtraData* objects to it).
Since these are the most complex classes, we will review them in more depth one by one, starting the **MouseClickTrackingBehavior**. Its implementation is displayed in Listing 4.14

Essentially, the MouseClickTrackingBehavior tracks all mouse clicks inside the window object (via a global click event) and uses *raycasting* (projecting an invisible line from the camera to the point of the click) to determine where exactly the click happened. After that, it checks if the intersect (the element hit by the raycast) is an object marked for tracking (in other words, object that has not been excluded from the array by the filters). If we determine that the user has indeed clicked on an object we are interested in, objects of type *InteractionData* and *ExtraData* are created and passed to the *onInteraction* callback function.

```
1
2  export class MouseClickTrackingBehavior extends TrackingBehavior {
3
4    public startTracking(
5      objects: any[],
6      references: { scene?: any; camera?: any; domElement?: HTMLCanvasElement;},
7      onInteraction: (data: InteractionData, extraData: ExtraData) => void): void
           {
8      this.objectsOfInterest = objects;
9      this.scene = references.scene;
10     this.camera = references.camera;
11
12     if (!this.scene || !this.camera) {
13       console.log("no scene/camera provided.");
14       return;
15     }
16
17     // Attach a click event listener
18     this.eventListener = (event: MouseEvent) => {
19       // Convert to normalized device coords, from -1 to 1
20
21       // raycast to know where the click happened
22       this.raycaster.setFromCamera(this.mouseVector, this.camera);
23
24       const threeMeshes = this.objectsOfInterest
25         .map(o => o.reference)
26         .filter((mesh: any) => mesh?.isMesh || mesh?.isObject3D);
27
28       const intersects = this.raycaster.intersectObjects(threeMeshes, false);
29
30       if (intersects.length > 0) {
31         const clickedMesh = intersects[0].object;
32         const foundObj = this.objectsOfInterest.find(o => o.reference ===
               clickedMesh);
33
34         if (foundObj) {
35           const interaction: InteractionData = {
36             objectId: foundObj.id,
37             userId: "", // filled in later on
38             eventType: "mouse click"
39           };
40
41           const extraData: ExtraData = {
42             mouseX: event.clientX,
```

```
43              mouseY: event.clientY,
44              timestamp: new Date().toISOString()
45            };
46
47            onInteraction(interaction, extraData);
48          }
49        }
50      };
51
52      window.addEventListener("click", this.eventListener);
53    }
54  }
```

Listing 4.14: MouseClickTrackingBehavior class - tracks all mouse clicks in the app but is only triggered if the click was on an object to track

Next, showcased in Listing 4.15, we have the **ControllerTrackingBehavior**. This class is responsible for monitoring user interactions with connected VR controllers. We again have the *startTracking* method which initiates the tracking process with the *onInteraction* callback function. First we get a reference to the two controllers and then use them to attach event listeners for the following events - *selectstart, selectend, squeezestart* and *squeezeend* (although only the pressing events are displayed in the code listing to avoid repetitive code). These events are provided by the *WebXR Device API*. As soon as one of them is triggered, corresponding *InteractionData* and *ExtraData* objects are created. We are tracking both the pressing of a button (selectstart and squeezestart events) and the releasing of that button (selectend and squeezeend events). This could be used to find out how long the user kept a button pressed, for example.

```

export class ControllerTrackingBehavior extends TrackingBehavior {

  public startTracking(
    objects: any[],
    references: {
      scene?: any;
      camera?: any;
      domElement?: HTMLCanvasElement;
      renderer?: THREE.WebGLRenderer;
    },
    onInteraction: (data: InteractionData, extraData: ExtraData) => void
  ): void {

    // attach event listeners to controller1 to monitor pressing and releasing
        of buttons
    if (this.controller1) {
      (this.controller1 as any).addEventListener("selectstart", () => {
        const interaction: InteractionData = {
          objectId: "controller1",
          userId: "",
          eventType: "controller button press"
        };

```

```
24          const extraData: ExtraData = {
25            timestamp: new Date().toISOString()
26          };
27
28          onInteraction(interaction, extraData);
29        });
30
31        (this.controller1 as any).addEventListener("squeezestart", () => {
32          const interaction: InteractionData = {
33            objectId: "controller1",
34            userId: "",
35            eventType: "controller button press"
36          };
37
38          const extraData: ExtraData = {
39            timestamp: new Date().toISOString()
40          };
41
42          onInteraction(interaction, extraData);
43        });
44      }
45
46      // Attach event listeners to controller2 to monitor pressing and releasing
          of buttons
47      if (this.controller2) {...}
48    }
49  }
```

Listing 4.15: ControllerTrackingBehavior class - tracks the pressing and releasing of buttons on the connected VR controllers

The Composer class

After that, we have the next pipeline component - the **Composers**. This class is responsible for transforming each tracked interaction into a structured format, in our case an xAPI statement. While our implementation uses the Experience API by default, developers can easily implement custom Composers if they need a different data format. Furthermore, to simplify the process of working with xAPI, we provide a chainable syntax in the form of *actor.does(verb).activity(activity)...*. This is achieved by using the *builder pattern* approach (for the chaining of the methods). To be able to do that, however, the **xAPIComposer** class also uses two "helper" classes - the *Actor* class and the *xAPIStatementBuilder* class. Essentially, the xAPIComposer class receives an *InteractionData* object from the *Tracking Behaviors*, inspects the type of the detected interaction (e.g. a mouse click) and initiates the process. It selects the appropriate verb and activity, takes the corresponding definitions provided by the *RWTH xAPI Registry*[5] and creates an instance of the *Actor* class with a username (defined by the developer) and, for now, a placeholder email. This Actor object is the starting point as it provides the *does(verb)* method. From there, an *xAPIStatementBuilder* instance with the respective verb is created. The builder object stores the actor and

[5] `https://xapi-stage.elearn.rwth-aachen.de/`, Accessed on March 22, 2025

the verb and provides the rest of the xAPI functions which return itself, thus enabling the chaining of methods (the aforementioned *builder pattern*). The components in the *Data Providers* section of the pipeline (see Figure 3.2 for clarity) will use this builder object as a container for the information (about the detected interaction) and as a way to communicate. For each statement, a new xAPIStatementBuilder object is created. Finally, the complete builder instance containing the core data about the xAPI statement (e.g. actor, verb and activity) is returned to be further modified by the next component in the pipeline. As defined in the base Composer class, Composers can also process a *string*. We offer this functionality in the event that a developer wishes to create custom components. A basic functionality flow could go as follows:

1. *Tracking Behavior* detects an event (e.g., a mouse click) and creates *InteractionData* and *ExtraData* objects.
2. The *`compose(InteractionData)`* method receives this *InteractionData*, inspects the interaction type, and selects the corresponding verb and activity definitions. It then constructs an `Actor` with username and email.
3. From the *Actor* instance we call the `does(verb)` method, returning an *xAPIStatementBuilder* instance.
4. The builder stores the core statement data and provides method chaining with further calls such as `.activity(...)` or `.withExtension(...)`.
5. At this stage, other pipeline components (e.g., hooks) can modify the builder to enrich the statement with additional metadata (e.g., mouse coordinates, timestamp).
6. Finally, an *Endpoint* calls the builder's `.build()` method, thus finalizing the xAPI statement which is now ready to be stored (e.g., in a Learning Record Store).

Parts of the implementation of the *xAPIComposer* are displayed in Listing 4.16 showcasing key functionalities of the statements' creation process. For the complete source code, please refer to the GitLab repository linked in the digital appendix.

```
export class xAPIComposer extends Composer {
  public compose(data: InteractionData): xAPIStatementBuilder {
    // pick correct verb and store it in chosenVerb
    // pick correct activity and store it in xapiActivity
    // create an Actor
    const actor = new Actor(data.userId, "testMail@test.com");

    // builder pattern
    const statementBuilder = actor.does(chosenVerb).activity(xapiActivity);

    // return the builder in case devs have implemented hooks
    return statementBuilder;
  }
}
```

Listing 4.16: xAPIComposer class - creates the structured xAPI statements with the help of the Actor and xAPIStatementBuilder classes.

The Hook class

Next, we have the **Hook** classes. Their role is to attach additional information to the statements, thereby making them more informative. To do that, we use *xAPI Extensions*. Currently, there are two prebuilt Hook classes - the **TimestampHook** and the **MousePositionHook**. As the names suggest, one adds the timestamp from the moment the detected interaction has occurred and the other the position of the mouse (in the case of a mouse click event). Each Hook operates in a similar way (as defined by the base class), i.e., they receive an *xAPIStatementBuilder* object from a *Composer* (or a string in the case of custom built composers) and an *ExtraData* object from a *TrackingBehavior* which contains the necessary metadata. Then, the Hooks call the *withExtension* method of builder object and pass to it the necessary metadata (e.g. `"builder.withExtension("mouseClickCoordinates", { extraData.mouseX, extraData.mouseY })"`). That information is then stored internally within the builder object which is again returned by the Hooks to be further modified by the next pipeline component (e.g., other hooks or an endpoint). An example implementation of a Hook class, showing the *MousePositionHook* is displayed in Listing 4.17.

```
1  export class MousePositionHook extends Hook {
2    public process(builder: xAPIStatementBuilder, extraData: ExtraData):
         xAPIStatementBuilder {
3      // if no coords, do nothing
4      if (extraData.mouseX === undefined || extraData.mouseY === undefined) {
5        return builder;
6      }
7
8      const mousePositionData = { x: extraData.mouseX, y: extraData.mouseY };
9      builder.withExtension("mouseClickCoordinates", mousePositionData);
10
11     return builder;
12   }
13 }
```

Listing 4.17: Hook implementations for modifying the xAPI statements via the xAPIStatementBuilder object

The Endpoint class

Lastly, there are the **Endpoint** classes. Essentially, their role is to finalize the statements and ensure their safe storage. Currently, there are two prebuilt Endpoint implementations - the **LrsEndpoint** (sends the statements to our Learning Record Store) and the **ConsoleEndpoint** (prints the statements out to the console, e.g., for testing purposes). Essentially, all endpoints expect to receive either a string or an xAPIStatementBuilder object. A string is accepted in the case of custom Endpoint classes that developers might be interested in implementing, but our pipeline uses the xAPIStatementBuilder class to create the statements. If a builder object is passed to the *send* method (their main method) of an Endpoint, it calls *build* method of that object which finalizes the statement with the information saved in it by the *Composers* and the *Hooks* that have previously processed it before storing it to the designated place (database, LRS, console etc.). Moreover, another key functionality can be found in the

build method. It is also used to format the final xAPI statements in the appropriate format, i.e., following the *TinCan* (Experience API). This is achieved via the *tincants* library which provides the *Agent, Verb, Activity and Statement* classes. In Figure 4.2 a display of a statement created via the tincants library is displayed. Otherwise, Endpoints also accept a string in case developers create custom components that pass around strings. An example Endpoint implementation is shown in Listing 4.18.

```
export class LrsEndpoint extends Endpoint {
  private lrs: LRS;

  constructor() {
    super();

    // LRS credentials
    this.lrs = new LRS({
      endpoint:***,
      username:***,
      password:***
    });
  }

  public async send(statementInput: string | xAPIStatementBuilder): Promise<
      void> {
    let statementJson: string;

    // 1) Convert to string if needed
    if (typeof statementInput === "string") {
      statementJson = statementInput;
    } else {
      statementJson = statementInput.build();
    }

    try {
      // 2) parse the JSON into an object
      const statementObj = JSON.parse(statementJson);

      // 3) create a Tincants Statement from it
      const finalStatement = new Statement(statementObj);

      // 4) send to the LRS
      const response = await this.lrs.saveStatement(finalStatement);
      console.log("Statement successfully sent to the LRS:", response);
    } catch (error) {
      console.error("Error sending statement to the LRS:", error);
    }
  }
}
```

Listing 4.18: Endpoint implementations for xAPI statement delivery

The Actor Pipeline and Data Providers classes

So far, we reviewed the six components of OmiLAXR's pipeline. The pipeline is, however, also split into two bigger "groups" - the **Actor Pipeline** and the **Data Providers** (see Figure 3.2). We have a class for each of the two in our "*OmiLAXR for WebXR*" implementation. The reason for that is to offer developers the option to create multiple instances of each one depending on their needs. This could be useful in a variety of situations, but a simple example would be having a virtual applications where users interact with NPC characters who are either friendly or hostile. The developer could have two different ActorPipeline instances tracking interactions with the non-player characters by, for example, using a *"Friend" tag* for one and an *"Enemy" tag* for the other. In this chapter, we will review both classes briefly one by one, starting with the **ActorPipeline** class, which is displayed in Listing 4.19.
It serves as a "manager" of the first three components in the pipeline. Developers use a *configuration object* to create an instance of the class. For that, we define an interface called "*ActorPipelineConfig*" in the *Pipeline* class which will be reviewed later in this chapter. Through that interface, developers can specify what components exactly they need by passing arrays with the *Listeners*, *Filters* and *TrackingBehaviors* they intend to use in their application. In that configuration object they also pass further arguments required for the tracking, like the *scene object* for example. The main functionality of the ActorPipeline class is defined in the *startActorPipeline* method, which receives the input parameters required for the three components and runs them in the correct order while also passing around data between them (e.g. the array of objects to track).

```
export class ActorPipeline {
  constructor(private config: ActorPipelineConfig) {
    this.listeners = config.listeners || [];
    this.filters = config.filters || [];
    this.trackingBehaviors = config.trackingBehaviors || [];

    this.scene = config.scene;
    this.camera = config.camera;
    this.domElement = config.domElement;
    this.renderer = config.renderer;
  }

  public startActorPipeline(
    scene: any,
    userID: string,
    onInteraction: (iData: InteractionData, eData: ExtraData) => void): void {
    // 1) Collect objects from each listener
    let allObjects: any[] = [];
    for (const l of this.listeners) {
      const scanned = l.scan(scene);
      allObjects.push(...scanned);
    }

    // 2) Pass through each filter
    for (const filter of this.filters) {
      allObjects = filter.apply(allObjects);
```

```
    }

    // 3) For each trackingBehavior, start tracking
    for (const tb of this.trackingBehaviors) {
      tb.startTracking(
        allObjects,
        {
          scene: this.scene,
          camera: this.camera,
          domElement: this.domElement,
          renderer: this.renderer
        },
        // callback
        (interactionData: InteractionData, extraData: ExtraData) => {
          // attach userID
          interactionData.userId = userID;

          // pass the data down the pipeline
          onInteraction(interactionData, extraData);
        }
      );
    }
  }
}
```

Listing 4.19: ActorPipeline class for coordinating listeners, filters, and tracking behaviors

Next, showcased in Listing 4.20, we have the **DataProviders** class. Similar to the ActorPipeline class, it "manages" the last three components in the pipeline. We again offer developers a configuration object for the initiation of a DataProviders instance, called *DataProvidersConfig*. The interface is also defined in the *Pipeline* class and accepts three arrays configuring the different implementations of the *Composers, Hooks* and *Endpoints* developers intend to use. The core functionality happens inside the *handleInteraction* method, which receives the instances of the two interfaces we previously explained from the *Actor Pipeline* components - the *InteractionData* object and the *ExtraData* object. Internally, the DataProviders class runs the three components with the data from the two objects in their respective order, managing the communication between them (by passing around the *xAPIStatementBuilder* object) and ultimately orchestrating the creation and storage of the statements.

```
export class DataProviders {
  constructor(private config: DataProvidersConfig) {
    this.composers = config.composers || [];
    this.hooks = config.hooks || [];
    this.endpoints = config.endpoints || [];
  }

  public handleInteraction(interactionData: InteractionData, extraData:
      ExtraData): void {
    // For each composer, compose a statement
    for (const composer of this.composers) {
      let statement = composer.compose(interactionData);

```

```
      // run the hooks
      for (const hook of this.hooks) {
        statement = hook.process(statement, extraData);
      }

      // endpoints send final statements to LRS/Console
      for (const endpoint of this.endpoints) {
        endpoint.send(statement);
      }
    }
  }
}
```

Listing 4.20: DataProviders class for managing composers, hooks, and endpoints

The Pipeline Class

Finally, showcased in Listing 4.21, there is the **Pipeline** class. It serves as the *central coordinator* and provides the *entry points* for developers to use the framework. Here we define the previously mentioned interfaces - *ActorPipelineConfig*, *DataProvidersConfig* and the *PipelineConfig*. The key configuration object here is the *PipelineConfig*. It accepts objects of the type of the other two interfaces and the necessary objects and variables for the framework to be able to work with the virtual application. To showcase this concept, a test application that uses the framework will be shown later in the paper. The functionality of this class is rather straightforward. First, in the constructor, we create the instances of the *ActorPipeline* and the *DataProviders* classes based on the given configurations. Then, via the *start* method, the entire pipeline is initialized. With its design, the data flow and communication between the different classes is done by chaining tasks from one component to the next one. Here, the callback function plays a key role as it bridges the two bigger parts of the pipeline - the *Actor Pipeline* and the *Data Providers*. We define that callback function in the Pipeline class. Essentially, every time an interaction is detected the callback starts the DataProviders instances and passes them the *InteractionData* and *ExtraData* objects so that they can create a statement describing the detected action.
A basic functionality flow would go like this:

1. Each *ActorPipeline* is started using the provided scene and user ID.
2. As user interactions are detected and transformed into *InteractionData* and *ExtraData* by the actor components, the provided callback is triggered.
3. The data about the detected interaction is sent to the *Data Providers* and a statement is created and stored or printed out in the console.

A slightly more detailed representation of this process can be seen in Figure 4.3 which aims to bring clarity to the project by visualizing a simplified functionality flow.

```
export class Pipeline {

  constructor(public config: PipelineConfig) {
    // create ActorPipeline instances from actorConfigs array
```

```
5       this.actorPipelines = config.actorConfigs.map(aCfg => {
6         return new ActorPipeline(aCfg);
7       });
8
9       // multiple DataProviders instances
10      this.dataProviders = config.dataProvidersConfig.map(dpCfg => {
11        return new DataProviders(dpCfg);
12      });
13    }
14
15    // starts each ActorPipeline
16    public start(scene: any, userID: string): void {
17
18      // For each actor pipeline, start it with a callback
19      for (const actorPipeline of this.actorPipelines) {
20        actorPipeline.startActorPipeline(scene, userID, (iData: InteractionData,
              eData: ExtraData) => {
21          // for each DataProviders instance
22          for (const dp of this.dataProviders) {
23            dp.handleInteraction(iData, eData);
24          }
25        });
26      }
27    }
28  }
```

Listing 4.21: Pipeline configuring ActorPipeline and DataProviders instances via the configuration interfaces and managing the frameworks functionality via the start method

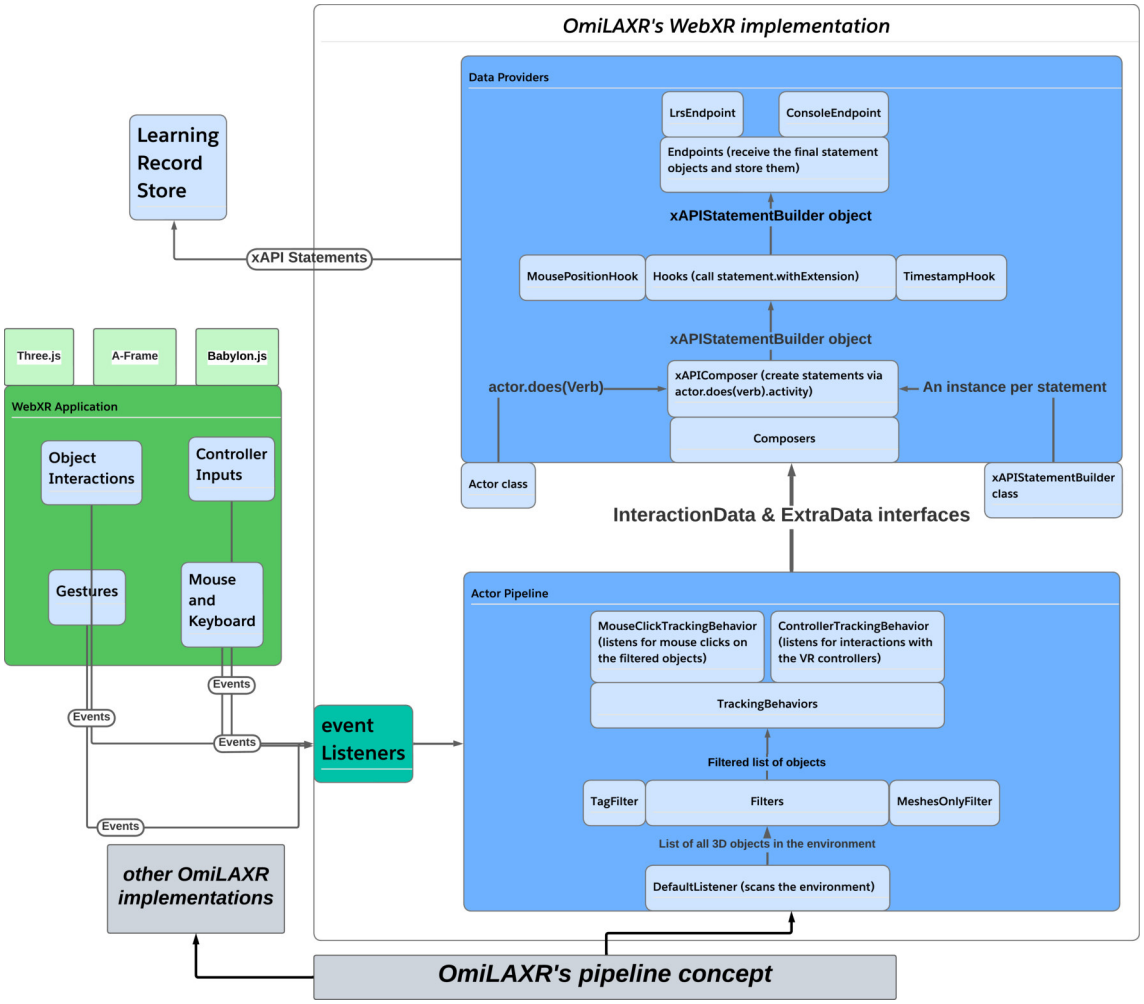

Figure 4.3: A high-level illustration of the OmiLAXR framework adapted for WebXR environments. The diagram highlights how interaction data is captured from user inputs (e.g., mouse, controller, gestures) within a WebXR application built using libraries like Three.js or A-Frame. These inputs trigger events processed by listeners, which then pass relevant objects through a filtering stage and into tracking behaviors. Once interactions are detected, the TackingBehaviors pass *InteractionData* and *ExtraData* objects to the Data Providers and xAPI statements are generated using the builder pattern (actor.does(verb).activity). Finally, statements are sent to an endpoint (e.g., an LRS). The image was created with the *Lucidchart*[6] web application by the author.

The code bundling process

By now, we have introduced the main classes of the project. There are, however, a few more files that enable the functionality, which we will briefly talk about. Since the final goal of the project is to produce a **JavaScript library**, we use **Rollup.js** to bundle the code. To make that process simpler, we have the **index.ts** file which exports every single class that developers using the framework might need and serves as the entry point for the bundling process. Rollup compiles and packages the entire project. For that, we create a configuration file (*rollup.config.js*) which starts with the *index.ts* file, resolves dependencies, compiles TypeScript into JavaScript and ultimately generates

an output in the formats we choose (an ES module bundle (omilaxrWebXR.esm.js) in our case). This makes the framework portable and easy to integrate.

4.3 Testing the framework

After introducing the framework properly, we can now take a look at how exactly it is used inside an actual WebXR environment. We will test it on two different Three.js applications - each testing a different tracking mechanism, starting with mouse clicks.

4.3.1 Tracking mouse clicks

This test application will be similar to the one used for the C# version. It has a plane object to represent a large white floor, black background and three cubes of different colors (green, red and blue). The cubes act as buttons and when clicked, the floor changes its color to the color of the clicked button. How the *bundled OmiLAXR library* is imported in the code is shown in Listing 4.22. As shown there, we import the components we plan on using and create the configuration objects. An interesting detail to note is, for example, the "object of interest" *tag* given to each cube so that we ensure to only track clicks performed on them directly. Screenshots of the application and the results (i.e. the resulting statement) are shown in Figure 4.4. An important detail to note here is the web links from the statement which represent the **xAPI URIs**. Once clicked, they lead to the definition page of the respective verb, activity etc. from the **xAPI Registry**. In this case, we have the verb **clicked** and the link leads to the page in the registry for that verb[7].

```
// import the necessary OmiLAXR components
import {
  Pipeline,
  DefaultListener,
  TagFilter,
  MouseClickTrackingBehavior,
  xAPIComposer,
  MousePositionHook,
  TimestampHook,
  ConsoleEndpoint
} from "./omiLaxrWebXR.esm.js";

// Three.js code for the creation of the virtual environment

// build the pipeline config
const pipelineConfig = {
  // write an array of actor configs:
  actorConfigs: [
    {
      listeners: [ new DefaultListener() ],
      filters: [ new TagFilter("object of interest") ],
      trackingBehaviors: [ new MouseClickTrackingBehavior() ],
      // references for the scene
      scene: scene,
```

[7] `https://xapi.elearn.rwth-aachen.de/definitions/generic/verbs/clicked`, Accessed on March 25, 2025

```
25        camera: camera,
26        domElement: renderer.domElement,
27        renderer: renderer
28      }
29    ],
30    dataProvidersConfig: [
31      {
32        composers: [ new xAPIComposer() ],
33        hooks: [ new MousePositionHook(), new TimestampHook() ],
34        endpoints: [ new ConsoleEndpoint() ]
35      }
36    ],
37    scene: scene,
38    camera: camera,
39    domElement: renderer.domElement,
40    renderer: renderer
41  };
42
43  const pipeline = new Pipeline(pipelineConfig);
44  // start the pipeline with user ID "testUser"
45  pipeline.start(scene, "testUser");
```

Listing 4.22: Example usage of the OmiLAXR pipeline in a Three.js app

4.3.2 Tracking controller buttons

In this subchapter, we are going to test the tracking of controller buttons. To do that, this time we will use a prebuilt application instead of a basic one developed by the author for testing purposes. This application is provided as an example of what can be created with Three.js on the framework's official website[8]. It is called *Ballshooter*[9]. Essentially, the app creates a room with walls and also virtual objects representing the connected VR controllers. When a button on the controllers is pressed, the virtual controllers shoot out balls that bounce around in the room. We are going to import our OmiLAXR library in that application to integrate tracking mechanisms in a similar manner as the code shown in Listing 4.22 with the only differences being that we are going to use the *ControllerTrackingBehavior* instead of the *MouseClickTrackingBehavior* and the *Filters* and *Listeners* arrays will be empty as we are not tracking any object interactions in this example (i.e. we are not interested in scanning or filtering any objects in this case). It should also be noted that this test is performed via a Chrome extension that simulates VR hardware due to the lack of physical devices. The extension is called *Immersive Web Emulator*[10]. The results are displayed in Figure 4.5. An interesting functionality to note is that we are tracking both the *pressing* and *releasing* of the controller buttons. A statement is created for each action. Again,

[8] `https://threejs.org/examples/`, Accessed on March 25, 2025
[9] `https://threejs.org/examples/webxr_xr_ballshooter.html`, Accessed on March 25, 2025
[10] `https://chromewebstore.google.com/detail/immersive-web-emulator/cgffilbpcibhmcfbgggfhfolhkfbhmik?hl=en`, Accessed on March 25, 2025

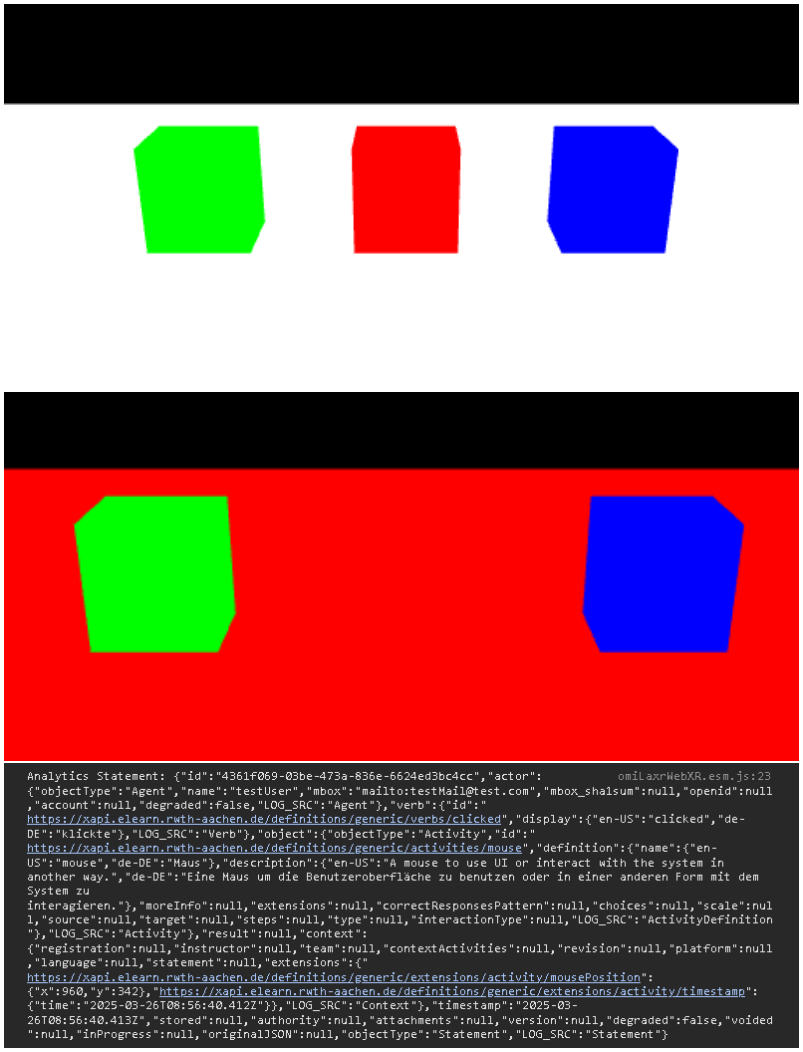

Figure 4.4: Screenshots showing a Three.js app that imports the OmiLAXR library. Initially, we see the starting position, then the result after having clicked on the red cube and finally the resulting xAPI statement. The images are generated by the author.

URIs for the xAPI elements are provided in the statement and for example the verb IDs lead to the web pages for the verbs *pressed*[11] and *released*[12].

4.3.3 Creating custom components

Another important development goal which was of high priority from the beginning is to design the framework in such a way that we make it easy for developers to create custom logic and feed it into the pipeline. Integrating custom logic allows developers to have much more freedom while using the framework and it could be very practical in a variety of use cases, such as custom TrackingBehavior classes that monitor some rare or very specific interactions, custom filtering mechanisms, using a different data format than the Experience API, custom Hooks or custom Endpoint classes

[11] `https://xapi.elearn.rwth-aachen.de/definitions/virtualReality/verbs/pressed`, Accessed on March 25, 2025

[12] `https://xapi.elearn.rwth-aachen.de/definitions/virtualReality/verbs/released`, Accessed on March 25, 2025

that store the statements in specialized databases or directly in a dashboard web application and many more. In the current design, the way developers would handle the creation of new components is very simple. They only need to import the base (abstract) class of each component for which they wish to write custom logic, create the class and pass an instance of it to the pipeline via the *ActorPipelineConfig* or the *DataProvidersConfig*. To showcase this functionality we present an implementation of a basic filter, called "NoMeshesFilter" - filters out all the *Meshes* from the arrays of objects. First, in our WebXR application we have to import the base (abstract) *Filter* class from the bundled OmiLAXR library and then write the new class as a derived class. To showcase the functionality, we provide an example in Listing 4.23.

```
export class NoMeshesFilter extends Filter {
  public apply(objects: any[]): any[] {
    return objects.filter(obj => obj.isMesh !== true);
  }
}
```

Listing 4.23: NoMeshesFilter implementation

Then, in order for the pipeline to actually use the filter, we need to pass an instance of it inside the ActorPipelineConfig object which itself is used in the *PipelineConfig* interface. What this could look like, is shown in Listing 4.24

```
actorConfigs: [
  {
    listeners: [ new DefaultListener() ],
    filters: [ new TagFilter("object of interest"), new NoMeshesFilter() ],
    trackingBehaviors: [ new MouseClickTrackingBehavior() ],
    // rest of the objects like scene or camera...
  }
]
```

Listing 4.24: Integrating custom filtering logic in the pipeline by simply feeding it to the configuration interface

4.3.4 Evaluating the framework

After presenting the technicalities of the project, we will briefly evaluate it in this chapter. **Disclaimer**: Given that the main goal of this thesis was to explore potential approaches to realize the OmiLAXR framework in WebXR environments and considering the time constraints and scope of the project, a comprehensive evaluation involving external developers was not possible. Had there been a proper evaluation, it would have focused on both the technical performance and developer experience. On the technical side, the goal would be to test whether tracking interactions in real time and sending out xAPI statements to a Learning Record Store (LRS) causes any noticeable impact on performance. Things like frame rate, CPU and memory usage, and possible network latency would be measured and compared between a version of the app with and without OmiLAXR. Then, from a developer point of view, a small group of users to test the framework and report on their experiences would have been created. Instead, the following section provides a review based on the author's own experiences during the development and testing phases of the project. Furthermore,

in chapter "*Future Work*", potential updates focused on extending the framework's functionality will be discussed.
In terms of *Evaluation Criteria*, we are going to briefly discuss how the project scores under the following criteria (based on the author's experience): "*Separation of concerns*" (Are responsibilities split clearly across classes?), "*Extensibility*" (How much effort is required to create custom logic?), "*Usability*" (how hard is it for developers to work with the framework?), "*Framework independence*" (Does the framework work as expected across different applications?), "*Scalability*" (How scalable is the design?), "*Challenges of the approach*" (What challenges have we identified during the development process?).

1. **Separation of concerns:** With the design of the TypeScript approach we achieve a certain level of "*separation of concerns*" as the responsibilities of the framework are separated between the components with each component having a specific task.

2. **Extensibility:** The framework achieves its goal of allowing developers a certain level of freedom in the creation of new components. As showcased in the previous sections, to extend the framework, developers only need to import the necessary base class and implement their custom logic as a derived class of that base class.

3. **Usability:** Designing the framework in a "*user-friendly*" way has been a key concern throughout the development of the project. For instance, by providing interfaces and abstract base classes, we ensured a more intuitive development experience for developers. Each core functionality is isolated in its respective pipeline component and designed to be easily extensible, and all logic is bundled into a library that developers can selectively import based on their requirements (i.e., only import the components they actually need). Nevertheless, some basic improvements are still possible. For example, identifying common use cases and providing built-in support for those (e.g., to reduce boilerplate code), implementing complex error-catching mechanisms or creating a documentation with usage examples.

4. **Framework independence:** While the presented project was mainly focused on the *Three.js* framework, support for other frameworks can be achieved. Essentially, certain framework-specific components for the *Actor Pipeline* part of the pipeline (e.g., listeners for A-Frame or Babylon.js) would need to be developed. The *Data Providers* section would be compatible right away, as it does not use any framework-specific functionalities.

5. **Scalability:** The project would have to be tested on a complex project to understand its ability to scale better. Nonetheless, the current design can be adapted to challenges like multiple users or scenes, for example, by creating multiple pipelines.

6. **Challenges of the approach:** In comparison to developing OmiLAXR for Unity, realizing it for a WebXR environment introduced certain challenges due to fundamental differences between the two platforms. In Unity, developers benefit from a visual editor, where OmiLAXR components could, for example, be added directly to GameObjects using the Inspector. In WebXR, we need to use a property like the custom tag to achieve a similar effect. Furthermore, as developers (in WebXR environments) usually work with JavaScript, for example, our version of the framework had to be initialized and connected entirely through code at runtime. As a result, for the WebXR version of OmiLAXR, we prioritized the creation of an importable library that provides the necessary functionality.

In conclusion to this section, we could state that this approach is well-suited and intuitive with respect to the original development requirements - realizing the core functionalities of the OmiLAXR framework in WebXR environments. Its extensible structure has been realized to a certain extent, as creating new components is relatively straightforward and requires minimal boilerplate. Furthermore, compliance with the xAPI standard was also achieved through integration of the *TinCan* library (for TypeScript), thereby addressing the issue of standardization. While improvements are certainly possible, this approach serves as a solid foundation for implementing the framework in a WebXR context.

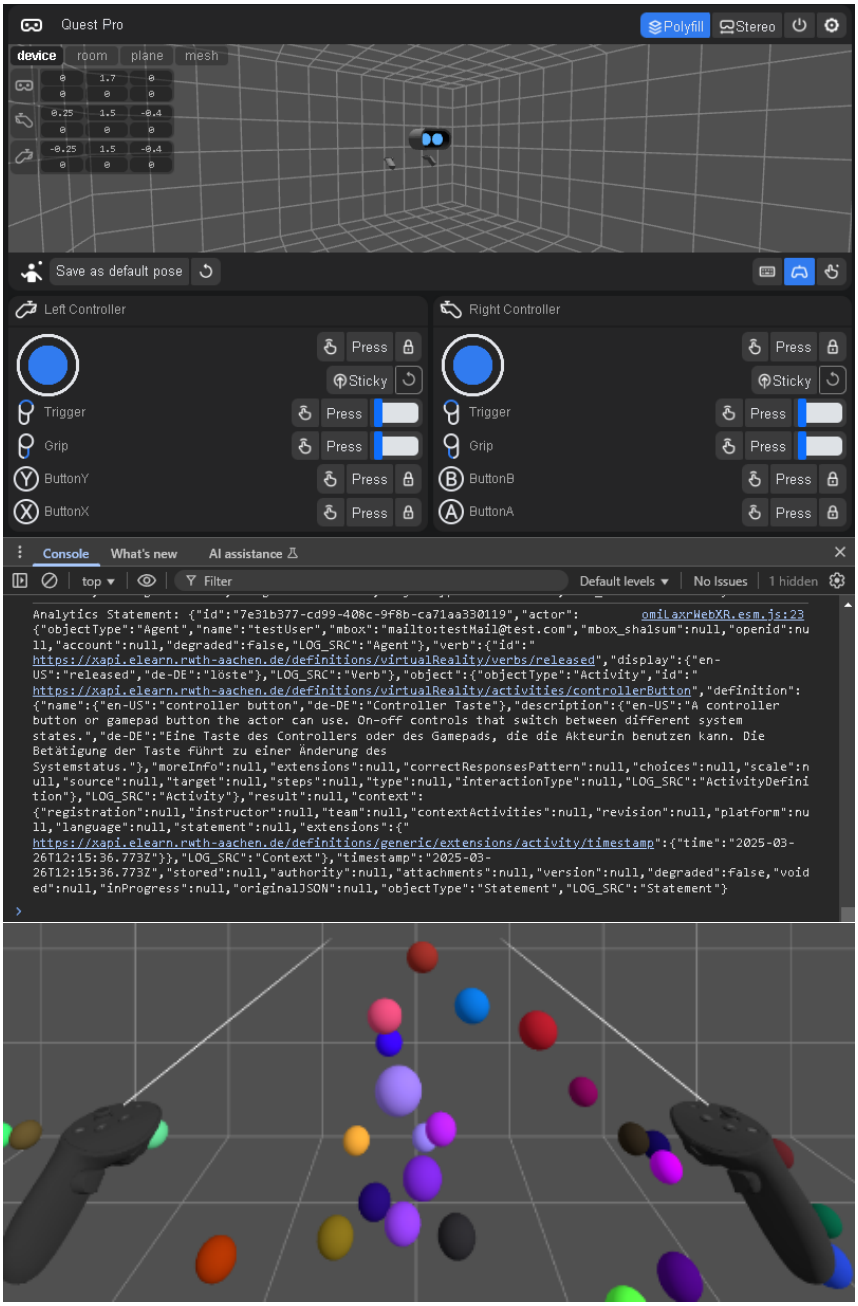

Figure 4.5: Screenshots of the Ballshooter application and the generated statement after pressing the button on a controller.
The *ballshooter* application was taken from the following website `https://threejs.org/examples/` (Accessed on March 25, 2025). The images are generated by the author.

Chapter 5 Future work

So far, we introduced OmiLAXR and the two approaches we developed for the *realization of the framework in WebXR environments*. Ultimately, we concluded that the second version, developed with TypeScript, was the better option and focused on it. In this chapter, we will talk about possible future directions for its development and also about problems or challenges that will should fixed in the near future, starting with the challenge of adapting the framework to work with other frameworks, like *Babylon.js* and *A-Frame*.

5.1 Adapting OmiLAXR to other WebXR frameworks

As far as the problem of using OmiLAXR with other frameworks goes, adaptions are only required in the *Actor Pipeline* part of the pipeline. First, we need to address the differences in the way each framework handles its objects. For instance, in Three.js, objects are structured in a tree-like format (see Figure 2.4 for clarity) and are children of the *scene*, whereas in Babylon.js we have a more direct access to the objects and can simply call `scene.meshes`, which returns an array of all the 3D Meshes (objects) in the scene. Then, in A-Frame, where applications are built in its HTML-based declarative approach, the objects are DOM elements (e.g., <a-box>, <a-entity>, etc.) and we would have to use a functionality like the *querySelector* to retrieve all the objects, as shown in Listing 5.1.

```
1 const sceneEl = document.querySelector('a-scene');
2 const entities = sceneEl.querySelectorAll('a-entity');
```

Listing 5.1: Querying A-Frame entities/objects

After retrieving the objects from the virtual environment, we must consider the differences between the frameworks that affect the filtering stage. For example, one key functionality we need to consider is giving developers the ability to "mark" interesting objects they wish to track (i.e., when users interact with them). For Three.js we provide the *TagFilter* that takes advantage of the *userData* property assigned to 3D objects, which is where developers define the *Tag*. There are similar functionalities in the other frameworks as well. For instance, in Babylon.js, there is the *metadata*[1] property that serves a similar purpose and can be used to define a tag. Then, in A-Frame, we can simply use something like a custom attribute (e.g. `<a-box object-tag="object of interest"></a-box>`). Afterwards, when it comes to the

[1] `https://doc.babylonjs.com/typedoc/classes/BABYLON.Mesh#metadata`, Accessed on March 27, 2025

actual pipeline components, we could simply implement them and include them in the pipeline. For example, we could have a *BabylonTagFilter* class that still extends the abstract base class Filter and filters based on the metadata object (e.g., via `if (mesh.metadata?.tag === "object of interest") ... )` or a *BabylonDefaultListener* that returns `scene.meshes`. Similarly, we could have an *AFrameTagFilter* that could work like shown below:

```
return objects.filter(el => el.getAttribute("object of interest") ===
this.tag);
```

Another interesting development idea could be useful, is to extract the tag already in the Listener, which is a basic string, and thus make the TagFilter universal across all frameworks.

Finally, the last classes that require adaptions for other frameworks are the *TrackingBehaviors*. Once again, due to the similarities between Three.js, Babylon.js, A-Frame etc. only small changes are needed. To clarify that, we will use the *MouseClickTrackingBehavior* class as an example. Essentially, the main functionality we need to adapt here is the *Raycasting* logic (used to check if the mouse click happened on an object we wish to track). For instance, in Babylon.js we could use predefined functionalities like the *Ray*[2] class, the *scene.pick(scene.pointerX, scene.pointerY)* method to perform the raycast, the returned *PickingInfo*[3] object, from which we get the object that was clicked on and so on. The rest is then very much the same as for all other MouseClickTrackingBehaviors, i.e. we check if the clicked object is an object we are tracking and trigger the callback function to create an xAPI statement if so. On the other hand, in A-Frame we could, for example, add "click" event listeners to the objects of interest.

Developers can then import only the necessary components for one framework depending on their application. Making that possible is part of the original goals we set before developing the framework, i.e., making the framework easily extensible (also the "Plug and Play" concept) for developers.

5.2 Smaller improvements

After briefly reviewing how OmiLAXR can be extended to support other frameworks (e.g., Babylon.js and A-Frame), this section will review a few smaller adjustments the framework could benefit from.

1. **Creation of a LocalEndpoint**: A LocalEndpoint class that saves the statements locally on the hard drive would make the framework more resilient as it would allow for the safe storage of the statements locally in the case of a problem with the Learning Record Store, for example. There are a few challenges associated with this concept. For one, applications running in the browser are *sandboxed* (for safety reasons), which prevents direct access to the user's hard drive (or file system). We could store the statements in the *localStorage*[4] object provided by the *window* interface. Nevertheless, we would still have to offer the user a way of downloading the stored statements (e.g., automatically after a number of

[2] `https://doc.babylonjs.com/typedoc/classes/BABYLON.Ray`, Accessed on March 28, 2025
[3] `https://doc.babylonjs.com/typedoc/classes/BABYLON.PickingInfo`, Accessed on March 28, 2025
[4] `https://www.w3schools.com/jsref/prop_win_localstorage.asp`, Accessed on March 28, 2025

statements has been reached or design a "download" button, etc.). Furthermore, we would have to research the cases for different browsers to avoid browser-specific issues, etc.

2. **Tracking more interactions and behaviors**: Offering additional TrackingBehavior classes that monitor a broader range of interactions is another key step in making the framework more practical and useful. We could develop tracking mechanisms for actions like hand gestures, gazing, voice commands and many more.

3. **Extending the results with a visual element**: Another interesting improvement we could develop for OmiLAXR is a tool that displays the collected information about user interactions in the virtual environment so that developers can directly see the effects of their XR applications (e.g., a dashboard that receives the xAPI statements and uses them to display certain graphics, statistics, etc.) [HGD+24].

4. **Implementing a *stop* method for the pipeline**: Currently, we initialize a pipeline instance and let it run in the background via the *start* method. Similarly, a *stop* method could be useful as well. For instance, if a developer creates multiple ActorPipeline instances and one is suddenly no longer needed (i.e. the objects it is tracking are no longer present in the environment), they could call the stop method. Such a method would at the very least remove event listeners, clear some memory (e.g. internal arrays or references to objects) or possibly allow developers to restart or reconfigure a given pipeline instance and so on.

5. **Using web workers**: Web workers can help improve the performance of OmiLAXR in different parts of the project while also helping keep the app responsive. We could for example use them for the listeners and filtering logic if we have a big application with hundreds or possibly thousands of objects in it or if we are composing an enormous number of statements and sending them all to an LRS.

Chapter 6 Conclusion

In this thesis, we explored how the concept of the **OmiLAXR** framework (Open and Modular Integration of Learning Analytics in XR) can be realized in WebXR environments. What OmiLAXR essentially does is it makes it easy for developers to integrate Learning Analytics mechanisms in their XR-based applications [GHS24b, GHS25]. This is achieved by scanning the virtual environment, monitoring interactions and experiences with the objects (e.g., grabbing an object) or the environment (e.g., teleportation), and then logging detailed information about the tracked actions in a standardized format (in our case xAPI statements) [GHS24b, GHS25]. The focus of the thesis revolved around the latest version of the framework, which is structured as a pipeline, where the entire functionality is separated into different components [GHS25]. Each component has its respective responsibility in the working process of OmiLAXR with the data being passed from one component to the next one in the pipeline [GHS25]. To realize this functionality in WebXR environments, two different approaches were implemented and reviewed in this paper. In both cases, the end product was a bundled JavaScript library that exports the necessary functionality to the WebXR application.
The first approach involved using C# for the core pipeline structure which is then compiled to WebAssembly so that it can be run in a web environment, and JavaScript for the communication with both the browser and the WebXR Device API. What we ultimately concluded, however, is that this approach did not offer the flexibility and advantages we were aiming for. It proved to be hard to scale, the creation (and integration) of custom logic or custom pipeline components was unnecessarily hard for developers and also a big part of the core functionality had to remain on the JavaScript side of the project as using C# for communication with a web environment proved to be hard. Ultimately, this solution did not meet the main requirements of simplifying the integration of tracking mechanisms and Learning Analytics in XR in a developer-friendly way.
The second approach was created with TypeScript as it provided the same benefits as JavaScript while also offering other advantages like type safety or type checking as well. This approach proved to be a better solution to the challenges of realizing OmiLAXR's functionality for WebXR environments. We again followed the pipeline design but achieved better separation of concerns (than with the first approach) as the code was not split between different programming languages. Furthermore, this approach also proved to be more developer-friendly as it was simple to use (a developer would only need to import a single file) and also made modifying the framework according to the developer's needs easier - developers can import only the components they need or define their custom functionality by importing and extending the base classes. Through these base classes creating custom logic is simple as they provide a

clear structure and also, by extending them, the framework accepts any custom-built logic.
As stated in the introduction, this thesis was motivated by the *research question* "*How can the OmiLAXR framework be realized in WebXR environments?*". To conclude the findings, we could state that after reviewing the two approaches, we perceived the best results by using TypeScript as a developing foundation. It allowed for a clear division of the responsibilities into separate components and adapting them to work with WebXR development technologies such as Three.js, Babylon.js, and A-Frame was successful. Furthermore, this design achieved other key goals set from the start - making the framework developer-friendly (to the possible extent) and giving developers a certain level of freedom in their usage of the project (i.e., creation and integration of custom logic into the framework is also doable).
Both approaches were tested on basic Three.js applications, where real-time tracking of *mouse clicks* was achieved by importing the bundled OmiLAXR library. The TypeScript version was also tested on an application provided by the official Three.js website.
In conclusion, we could state that this work provides a foundation for realizing the OmiLAXR framework in WebXR environments. The project can definitely benefit from certain improvements (e.g., the ones discussed in the "Future Work" chapter) but the initial testing results can be viewed as positive, aiming to increase the possibility of extending OmiLAXR for different XR development platforms.

Appendix A Bibliography

[AAW11] Abdul-Hadi G Abulrub, Alex N Attridge, and Mark A Williams. Virtual reality in engineering education: The future of creative learning. In *2011 IEEE global engineering education conference (EDUCON)*, pages 751–757. IEEE, 2011.

[ACISIAA22] Ahmed Alnagrat, Rizalafande Che Ismail, Syed Zulkarnain Syed Idrus, and Rawad Mansour Abdulhafith Alfaqi. A Review of Extended Reality (XR) Technologies in the Future of Human Education: Current Trend and Future Opportunity. *Journal of Human Centered Technology*, 1(2):81–96, August 2022.

[AII21] Ahmed Jamah Ahmed Alnagrat, Rizalafande Che Ismail, and Syed Zulkarnain Syed Idrus. Extended Reality (XR) in Virtual Laboratories: A Review of Challenges and Future Training Directions. *Journal of Physics: Conference Series*, 1874(1):012031, May 2021.

[Alh16] Wadee Alhalabi. Virtual reality systems enhance students' achievements in engineering education. *Behaviour & Information Technology*, 35(11):919–925, November 2016.

[ALR20] Carlos Alves and José Luís Reis. The intention to use e-commerce using augmented reality-the case of ikea place. In *Information Technology and Systems: Proceedings of ICITS 2020*, pages 114–123. Springer, 2020.

[BAA+22] Shrenik Bhansali, Ahmet Aris, Abbas Acar, Harun Oz, and A. Selcuk Uluagac. A First Look at Code Obfuscation for WebAssembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 140–145, San Antonio TX USA, May 2022. ACM.

[BMCC13] L. Bacon, L. MacKinnon, A. Cesta, and G. Cortellessa. Developing a smart environment for crisis management training. *Journal of Ambient Intelligence and Humanized Computing*, 4(5):581–590, October 2013.

[BW16] Paulo Blikstein and Marcelo Worsley. Multimodal learning analytics and education data mining: Using computational technologies to measure complex learning tasks. *Journal of learning analytics*, 3(2):220–238, 2016.

[CAH24] Nathaniel W. Cradit, Jacob Aguinaga, and Caitlin Hayward. Surveying the (Virtual) Landscape: A scoping review of XR in postsecondary learning environments. *Education and Information Technologies*, 29(7):8057–8077, May 2024.

[CDST12] Mohamed Amine Chatti, Anna Lea Dyckhoff, Ulrik Schroeder, and Hendrik Thüs. A reference model for learning analytics. *International Journal of Technology Enhanced Learning*, 4(5/6):318, 2012.

[CE23] Marcus Carter and Ben Egliston. What are the risks of virtual reality data? learning analytics, algorithmic bias and a fantasy of perfect data. *New media & society*, 25(3):485–504, 2023.

[CPL20] Athanasios Christopoulos, Nikolaos Pellas, and Mikko-Jussi Laakso. A Learning Analytics Theoretical Framework for STEM Education Virtual Reality Applications. *Education Sciences*, 10(11):317, November 2020.

[D+14] Jos Dirksen et al. *Three. js essentials*. Packt Publishing, 2014.

[Ded09] Chris Dede. Immersive Interfaces for Engagement and Learning. *Science*, 323(5910):66–69, January 2009.

[DMAPS21] Joao De Macedo, Rui Abreu, Rui Pereira, and Joao Saraiva. On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 255–262, Melbourne, Australia, November 2021. IEEE.

[DUKS18] Christian Dibbern, Manuela Uhr, Dennis Krupke, and Frank Steinicke. Can webvr further the adoption of virtual reality? In *Mensch und computer 2018-usability professionals*, pages 377–384. Gesellschaft für Informatik eV Und German UPA eV, 2018.

[EHL+20] Matthias Ehlenz, Birte Heinemann, Thiemo Leonhardt, René Röpke, Vlatko Lukarov, and Ulrik Schroeder. Eine forschungspraktische perspektive auf xapi-registries. In *DELFI 2020–Die 18. Fachtagung Bildungstechnologien der Gesellschaft für Informatik eV*, pages 331–336. Gesellschaft für Informatik eV, 2020.

[GGL21] Xingrong Guo, Yiming Guo, and Yunqin Liu. The Development of Extended Reality in Education: Inspiration from the Research Literature. *Sustainability*, 13(24):13776, December 2021.

[GHS23] Sergej Görzen, Birte Heinemann, and Ulrik Schroeder. Ein konzept zur evaluierung eines ökosystems für die integration von learning analytics in virtual reality. In *21. Fachtagung Bildungstechnologien (DELFI)*, pages 311–312. Gesellschaft für Informatik eV, 2023.

[GHS24a] Sergej Görzen, Birte Heinemann, and Ulrik Schroeder. Evaluation of a framework for the integrating of Learning Analytics in Virtual Reality. In *Proceedings of DELFI Workshops 2024*, pages 10–18420. Gesellschaft für Informatik eV, 2024.

[GHS24b] Sergej Görzen, Birte Heinemann, and Ulrik Schroeder. Towards using the xAPI specification for Learning Analytics in Virtual Reality. In *LAK Workshops*, pages 260–271, 2024.

[GHS25] Sergej Görzen, Birte Heinemann, and Ulrik Schroeder. A generalized and modular pattern for a systematic integration of Learning Analytics in eXtended Reality. In *LAK Workshops*, 2025.

[Gör25a] Sergej Görzen. Overview of the omilaxr pipeline, 2025. Accessed: March 9, 2025.

[Gör25b] Sergej Görzen. Website of the omilaxr framework, 2025. Accessed: March 9, 2025.

[HEGS22] Birte Heinemann, Matthias Ehlenz, Sergej Görzen, and Ulrik Schroeder. xAPI Made Easy: A Learning Analytics Infrastructure for Interdisciplinary Projects. *International Journal of Online and Biomedical Engineering (iJOE)*, 18(14):99–113, November 2022.

[HGD+24] Birte Heinemann, Sergej Görzen, Ana Dragoljic, Lars Meiendresch, Marc Troll, and Ulrik Schroeder. A Learning Analytics Dashboard to Investigate the Influence of Interaction in a VR Learning Application. In *LAK Workshops*, pages 251–259, 2024.

[HLP21] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the web conference 2021*, pages 2696–2708, 2021.

[HRS+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, Barcelona Spain, June 2017. ACM.

[Hyt23] Boris Hyttinen. Interactive Web Application Development with .NET and WebAssembly. 2023.

[IGG+10] María Blanca Ibanez, José Jesús García, Sergio Galán, David Maroto, Diego Morillo, and Carlos Delgado Kloos. Multi-user 3d virtual environment for spanish learning: A wonderland experience. In *2010 10th IEEE International Conference on Advanced Learning Technologies*, pages 455–457. IEEE, 2010.

[IMEH16] AELB Idrissi, Mohamed Larbi BEN MAATI, and Ismail EL HADDIOUI. xapi: succeeding scorm as the new efficient standard for learning management systems. In *Conference Paper November*, 2016.

[Joh21] Julia Johansson. Performance and ease of use in 3d on the web: Comparing babylon. js with three. js, 2021.

[JPBG19] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, 2019.

[KN18] Axel Karlsson and Oscar Nordquist. Babylonjs and three. js: Comparing performance when it comes to rendering voronoi height maps in 3d, 2018.

[KR16] Jonathan M Kevan and Paul R Ryan. Experience api: Flexible, decentralized and activity-centric data collection. *Technology, knowledge and learning*, 21:143–149, 2016.

[LC25] Georgios Lampropoulos and Nian-Shing Chen. Assessing the educational impact of extended reality applications: Development and validation of a holistic evaluation tool. *Education and Information Technologies*, February 2025.

[MS18] Blair Maclntyre and Trevor F. Smith. Thoughts on the Future of WebXR and the Immersive Web. In *2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*, pages 338–342, Munich, Germany, October 2018. IEEE.

[MWJR19] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 11543, pages 23–42. Springer International Publishing, Cham, 2019.

[NPNP17] Srushtika Neelakantam, Tanay Pant, Srushtika Neelakantam, and Tanay Pant. Introduction to a-frame. *Learning Web-based Virtual Reality: Build and Deploy Web-based Virtual Reality Technology*, pages 17–38, 2017.

[PGG+24] Radu Emanuil Petruse, Valentin Grecu, Maja Gakić, Jorge Martin Gutierrez, and Daniel Mara. Exploring the Efficacy of Mixed Reality versus Traditional Methods in Higher Education: A Comparative Study. *Applied Sciences*, 14(3):1050, January 2024.

[RB17] Micha Reiser and Luc Bläser. Accelerate javascript applications by cross-compiling to webassembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 10–17, 2017.

[RG23] Glen Ross and Andrew Gilbey. Extended reality (xr) flight simulators as an adjunct to traditional flight training methods: a scoping review. *CEAS Aeronautical Journal*, 14(4):799–815, 2023.

[SA24] Hanan Sharif and Amara Atif. The Evolving Classroom: How Learning Analytics Is Shaping the Future of Education and Feedback Mechanisms. *Education Sciences*, 14(2):176, February 2024.

[SBG+24] Frederic Salmen, Martin Breuer, Sergej Görzen, Malte Persike, and Ulrik Schroeder. FAIR Learning Technologies with Web Components and Packages. In *Proceedings of DELFI 2024*. Gesellschaft für Informatik eV, Gesellschaft für Informatik e.V., 2024.

[SDS22] Tawseef Ayoub Shaikh, Tabasum Rasool Dar, and Shabir Sofi. A data-centric artificial intelligent and extended reality technology in smart healthcare systems. *Social Network Analysis and Mining*, 12(1):122, 2022.

[SHH18] Prabhjot Sandhu, David Herrera, and Laurie Hendren. Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in javascript and webassembly. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 1–13, 2018.

[Sie13] George Siemens. Learning analytics: The emergence of a discipline. *American Behavioral Scientist*, 57(10):1380–1400, 2013.

[SMY+21] Dmytro S Shepiliev, Ye O Modlo, Yu V Yechkalo, Viktoriia V Tkachuk, Mykhailo Mintii, Iryna S Mintii, Oksana Markova, Tetiana V Selivanova, Olena M Drashko, Olga O Kalinichenko, et al. Webar development tools: An overview. In *Proceedings of the 3rd Workshop for Young Scientists in Computer Science & Software Engineering (CS&SE@ SW 2020) Kryvyi Rih, Ukraine, November 27, 2020*, number 2832, pages 84–93. CEUR Workshop Proceedings, 2021.

[SSCD23] Ramteja Sajja, Yusuf Sermet, David Cwiertny, and Ibrahim Demir. Integrating AI and Learning Analytics for Data-Driven Pedagogical Decisions and Personalized Interventions in Education. *arXiv preprint arXiv:2312.09548*, 2023.

[SVPG16] Monica Sebillo, Giuliana Vitiello, Luca Paolino, and Athula Ginige. Training emergency responders through augmented reality mobile interfaces. *Multimedia Tools and Applications*, 75(16):9609–9622, August 2016.

[TDO+24] Janko Tufegdžić, Matija Dodović, Mihajlo Ogrizović, Nikola Babić, Jovan Đukić, and Dražen Drašković. Application of WebAssembly Technology in High-Performance Web Applications. In *2024 11th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN)*, pages 1–6, Nis, Serbia, June 2024. IEEE.

[TTFA24] Esmaeel Toni, Elham Toni, Mahsa Fereidooni, and Haleh Ayatollahi. Acceptance and use of extended reality in surgical training: an umbrella review. *Systematic Reviews*, 13(1):1–28, 2024.

[Ver24] Awa Vernyuy. Impact of Technological Advancements on Human Existence. *International Journal of Philosophy*, 3(2):54–66, May 2024.

[VESN+17] Noah Van Es, Quentin Stievenart, Jens Nicolay, Theo D'Hondt, and Coen De Roover. Implementing a performant scheme interpreter for the web in asm.js. *Computer Languages, Systems & Structures*, 49:62–81, September 2017.

[WCD+24] Fridolin Wild, Tim Coughlan, Sarah Davies, Rebecca Shepherd, Trevor Collins, Aitch King, and Jade Matos Carew. eXtended Reality and Accessibility in Online and Distance Learning: Exploring the Opportunities and Challenges. *Immersive Learning Research-Academic*, pages 153–163, 2024.

[WRP+19] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-wasm: Type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, January 2019.

[YLF+23] Geng Yu, Chang Liu, Ting Fang, Jinyuan Jia, Enming Lin, Yiqiang He, Siyuan Fu, Long Wang, Lei Wei, and Qingyu Huang. A survey of real-time rendering on web3d application. *Virtual Reality & Intelligent Hardware*, 5(5):379–394, 2023.

[YSD+10] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, January 2010.

[YTZ+21] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 533–549, Virtual Event, November 2021. ACM.

Appendix B Digital Appendix

The digital appendix is submitted digitally. To get access to this material they shall contact the author or LuFG i9. The project is also uploaded to the following GitLab repository: `https://gitlab.com/learntech-rwth/omilaxr-ecosystem/webxr`.

Eidesstattliche Versicherung
Declaration of Academic Integrity

Bekyarov, Alben
Name, Vorname/Last Name, First Name

378220
Matrikelnummer (freiwillige Angabe)
Student ID Number (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare under penalty of perjury that I have completed the present paper/bachelor's thesis/master's thesis* entitled

Realisierung des OmiLAXR Frameworks in WebXR Umgebungen

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt; dies umfasst insbesondere auch Software und Dienste zur Sprach-, Text- und Medienproduktion. Ich erkläre, dass für den Fall, dass die Arbeit in unterschiedlichen Formen eingereicht wird (z.B. elektronisch, gedruckt, geplottet, auf einem Datenträger) alle eingereichten Versionen vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without unauthorized assistance from third parties (in particular academic ghostwriting**.** I have not used any other sources or aids than those indicated; this includes in particular software and services for language, text, and media production. In the event that the work is submitted in different formats (e.g. electronically, printed, plotted, on a data carrier), I declare that all the submitted versions are fully identical. I have not previously submitted this work, either in the same or a similar form to an examination body.

Aachen, April 7, 2025
Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen/Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
§ 156 StGB (German Criminal Code): False Unsworn Declarations
Whosoever before a public authority competent to administer unsworn declarations (including Declarations of Academic Integrity) falsely submits such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment for a term not exceeding three years or to a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
§ 161 StGB (German Criminal Code): False Unsworn Declarations Due to Negligence
(1) If an individual commits one of the offenses listed in §§ 154 to 156 due to negligence, they are liable to imprisonment for a term not exceeding one year or to a fine.
(2) The offender shall be exempt from liability if they correct their false testimony in time. The provisions of § 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, April 7, 2025
Ort, Datum/City, Date

Unterschrift/Signature

Acknowledgement

Thank God, this is over :)