Todo list

Bei Abgabe: Anweisung nocite in Bachelorarbeit.tex entfernen

Bei Abgabe: In Bachelorarbeit.tex und Expose.tex Dokumentenoption overfullrule entfernen und die Option final eintragen

Bei Abgabe: In Packages.tex beim Package todonotes die Option disable eintragen, um Todos zu deaktivieren

Bei Abgabe: In Packages.tex beim TODO die größte Seitennummer eintragen

Wenn eine Version der Arbeit erstellt wird, die gedruckt werden soll in Packages.tex beim Package hyperref die Option urlcolor=blue entfernen

TECHNIK
HOCHSCHULE MAINZ
UNIVERSITY OF
APPLIED SCIENCES

Masterarbeit

zur Erlangung des akademischen Grades Master of Engineering
im Studiengang Geoinformatik und Vermessung

Sensorama: A task-specific Telepresence Application for Contemporary Dance

Hochschule Mainz

Fachbereich Technik

Vorgelegt von:	Vorname Nachname
	Straße
	PLZ Ort
	Matrikel-Nr. 123456
Betreut von:	Prof. Dr. Betreuer
Eingereicht am:	01.01.1970

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit

> **Sensorama: A task-specific Telepresence Application for Contemporary Dance**

selbstständig und ohne fremde Hilfe angefertigt habe. Ich habe dabei nur die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt.

Zudem versichere ich, dass ich weder diese noch inhaltlich verwandte Arbeiten als Prüfungsleistung in anderen Fächern eingereicht habe oder einreichen werde.

Mainz, den 01.01.1970 Vorname Nachname

Abstract

This study investigates the feasibility of creating a fully customised and task-specific telepresence application based exclusively on open web standards and free software for contemporary dance practice. It surveys existing technologies and paradigms and establishes a practical reference implementation to evaluate its basic functionality and the development process. The study arrives at a positive assessment of the existing technological landscape and the feasibility of producing task-specific web applications as an intrinsic component in smaller interdisciplinary projects with a strong focus on digital practice. It concludes with a proposal for an alternative development method termed 'code composting', describing a cyclical process of intuitive composition and analytical decomposition.

Keywords: telepresence, contemporary dance, motion capture, arts, open-source, computer science, software engineering

Kurzzusammenfassung

Diese Studie untersucht die Machbarkeit der Entwicklung einer aufgabenspezifischen Telepräsenzanwendung für den Einsatz im zeitgenössischen Tanz und basierend auf offenen Standards. Es wird ein Überblick über existente Technologien erabeitet und eine Referenzimplementierung erstellt, deren Entwicklungsprozess und abstrakte Funktion evaluiert wird. Die Studie gelangt zu einer positiven Einschätzung im Hinblick auf

die technologischen Möglichkeiten und die Praktikabilität einer fallspezifischen Softwareimplementierung als Teil von interdisziplinären Projekten. Sie schlägt den Begriff der ‘Code-Kompostierung’ vor, eine alternative Entwicklungsmethode bezeichnend, die sich auf zyklische Prozesse von intuitiver Komposition und analytischer Dekomposition stützt.

Schlagwörter: Open-Source, Telepräsenz, Zeitgenössischer Tanz, Bewegungserfassung, Kunst, Informatik, Softwareentwicklung

Contents

List of figures

List of tables

List of abbreviations

2D 2-dimensional

3D 3-dimensional

ADSL asynchronous digital subscriber line

AJAX asynchronous JavaScript and XML

API application programming interface

BVH Biovision hierarchy

CLI command-line interface

CPU central processing unit

CNCF Cloud Native Computing Foundation

CRUD create, retrieve, update and delete

CSS Cascading Style Sheets

DIY do-it-yourself

ES6 ECMAScript 6

GATT Generic Attribute Profile

GB gigabyte

GIS geographic information system

HRTF head-related transfer function

HTML HyperText markup language

HTTP HyperText transmission protocol

I/O input/output

I2C Inter-Integrated Circuit

IETF Internet Engineering Task Force

IMU inertial measurement unit

JGU Johannes Gutenberg University Mainz

JS JavaScript

JSON JavaScript Object Notation

JSX JavaScript XML

KB/s kilobyte per second

km kilometre

LE little-endian

MB megabyte

Mbit megabit

MCU multipoint control unit

ML machine learning

ms millisecond

NoSQL Not-only SQL

NPM Node Package Manager

NTP Network Time Protocol

OCI Open Container Initiative

OOP object-oriented programming

OS operating system

P2P peer-to-peer

PWA progressive web application

RAM random access memory

RFC request for comments

RTC real-time communication

SATC Software Assurance Technology Center

SDK software development kit

SFU selective forwarding unit

SOFA spatially oriented format for acoustics

SPA single-page application

SSD solid-state drive

SSL secure sockets layer

SQL structured query language

TCP transfer control protocol

TB terabyte

TS TypeScript

UDP user datagram protocol

UI user interface

UML unified modeling language

VR virtual reality

W3C World Wide Web Consortium

WebRTC web real-time communication

WebXR web mixed reality

WHATWG Web Hypertext Application Technology Working Group

List of listings

1. Introduction

1.1. Background

Remote collaboration has become increasingly prevalent in various professional environments through broader digitalisation efforts and significantly accelerated during the COVID-19 pandemic. As a result, teleconferencing and telepresence platforms that were initially used primarily for international business relations are now much more common in many work environments. These technologies allow people to work together remotely in real time, usually focusing on streaming video and audio, document sharing and collaborative whiteboarding. While this covers most use cases in desk-based workplaces, it lacks the immersive qualities required for practices such as contemporary dance, where people relate to physical presence and shared space. This became apparent in March 2020, when dancers could no longer rehearse and work together due to the lockdown. Despite this, there were attempts at using videoconferencing to stream and record collaborative rehearsals or dance classes. Still, these were confined to a screen-centric interface and limited to audio and video.

While commercial conferencing tools dominate in popularity among conferencing applications (Brandl, 2023), there are several free and open-source alternatives. However, these all focus on the most basic form of screen-based conferencing. Various domain-specific solutions for specialised applications, mainly in telemedicine, industry and the military, support more immersive remote collaboration. Still, these are task-specific and difficult to afford for smaller creative or artistic project setups.

Support for web standards is driven by key industry players (Davis et al., 2023), making a wide range of basic functionality accessible in web browsers, as well as access to display and sensor technology and cross-platform deployment on desktop and mobile devices. This development now provides an increased potential for smaller and more task-specific applications to be built and deployed relatively quickly. It opens up new possibilities for niche cases of remote collaboration, such as dance practice, where the collaborative agency could be extended from a composite of video streams to the creation of shared virtual environments that facilitate a more personal form of mediating a sense of shared presence.

The standard for real-time communication (RTC) in Browsers or web real-time communication (WebRTC) ('WebRTC: Real-Time Communication in Browsers', 2023) was first proposed by Google in 2011 and became an official World Wide Web Consortium (W3C) standard in 2021 (Couriol, 2021). It has become the basis for numerous applications, such as some of the conferencing tools mentioned above, media streaming servers such as Wowza or Ant, or real-time frameworks and servers such as Mediasoup, Janus or LiveKit. In its most basic form, WebRTC establishes peer-to-peer connections between devices, allowing low-latency exchange of media streams and arbitrary messages over data channels. However, it can accommodate other more complex and versatile scenarios.

1.2. Proposal

The proposed study examines the feasibility of creating a customised telepresence experience that explicitly covers a specific task not provided by common platforms or products. A potential target audience for such an application would be tiny and hardly warrants a commercial strategy of external product development, marketing and support. Extensive software development budgets are also rare in funding schemes supporting smaller cultural production endeavours, and it is relatively common for practitioners

themselves to dabble in experimental development or to have a creative coder on the team. To keep the budgetary requirements for such an implementation at a minimum, relying on open standards and non-proprietary components is imperative. While the implementation has to fulfil a particular task, some level of abstraction, modular composition and separation of concerns are important design factors that allow for establishing a technological base that can be reused in multiple contexts with less work in subsequent deployment instances.

To support a broad range of scenarios, the application core should support the real-time streaming of any type of sensor data in addition to the usual video and audio streams. This would allow augmenting the telepresence environment with spatial data, sensor readings or generative data sources. The data could then be streamed as is but visualised, sonified, or otherwise analysed and processed on the receiving devices as required by the implemented use case. In this particular example, movement sonification is implemented as an alternative to the visually-centred conferencing paradigms. As movement in front of a screen or with headsets can be somewhat limiting, the idea is to provide spatial audio as a medium for verbal communication, transmission of audible representations of movement and a sense of positional orientation in relation to the virtual presence in the space. Focused on a scenario of two participants moving at remote locations but in virtually overlayed spatial dimensions, this setup could enable exploration of moving together by attempting to achieve some form of acoustic harmony or rhythm to supplant the lack of an actual shared physical presence. This implementation targets only a small audience in that it requires practice and a deep engagement with the sonification method, as it would be specifically built to express a specific style of movement that would not be intuitive for every potential user alike. It should also be used by dancers with a shared experience of moving together so that verbal communication can support navigating a shared movement vocabulary and connect to the memory of shared physical practice. Creative design processes and user experience are deemed outside the scope of this study, as the focus lies on examining the general feasibility and affordability of using open standards and free software to enable the creation of a task-specific real-time application,

exemplified by the proposed example and examined in its general functionality instead of usability or user experience.

1.3. Method

The study starts from a survey of *conceptual foundations* presenting existing paradigms and technologies to support the development of web-based real-time applications, establishes a *methodology* and presents an *application concept* as well as the resulting reference implementation. The reference implementation is the basis for a quantitative *evaluation* of its general functionality and a *critical reflection* on the development process. The study concludes with a general recommendation on the feasibility of creating such a 'single-use' application for a specific task and an outlook for future implications and possibilities resulting from this research.

2. Conceptual foundations

2.1. Telepresence

The term *telepresence* first appears in an article by Marvin Minsky, roughly defined as a form of remote robotic operation, that 'emphasises the importance of high-quality sensory feedback' and the author posits that its realisation's biggest challenge is 'achieving that sense of "being there."' (Minsky, 1980). Considering the general technological development at that time, Minsky argued from a standpoint concerned primarily with robotic manipulators that perform remote labour, enhancing the operator's physical abilities and safety. Today, virtual and augmented reality, telepresence, and general presence research present much more diverse application scenarios. While there are applications of remote robotic control in retail, industry, telemedicine and police or military, the most common instance has become the teleconferencing application relaying video and audio streams and allowing chat and collaborative whiteboards.

In this study, the term telepresence is used to explicitly describe a virtual or augmented environment that allows multiple people to experience some form of shared presence, immersion and interaction. The concepts of *presence* and *immersion* require a precise contextualisation for this study. The article 'A Survey of Presence and Related Concepts' (Skarbez et al., 2017) presents a wide range of possible variations and specific definitions for these concepts in different contexts and environments. There, the authors state that presence 'is most commonly defined as something akin to the feeling of "being there" in a virtual place' (Skarbez et al., 2017, p. 2) and immersion can be understood as 'an objective

characteristic of a [virtual environment] system' that, as the authors are citing Slater, 'provides the boundaries within which [presence] can occur' (Skarbez et al., 2017, p. 3). The goal, in this case, is not to transport the participants to a different virtual place but rather to make them experience the virtual presence of the other in the physical location they are in. Furthermore, there is no immersion in the usual sense as the participants can remain aware of the divide between the physical and the virtual, much like in a telephone conversation. The survey presents various definitions for mediated interaction in virtual environments and distinguishes '*copresence* as the sense of being together with another or others, and *social presence* as the moment-by-moment awareness of the copresence of another sentient being accompanied by a sense of engagement with them.' (Skarbez et al., 2017, p. 4). These terms are further differentiated as *social presence illusion* referring 'to the feeling of social presence engendered by characters in virtual or mediated environments' (Skarbez et al., 2017, p. 4) and *copresence illusion* as referring 'to the feeling of "being together" in a virtual or mediated space' (Skarbez et al., 2017, p. 5). The authors also note that the experience of these forms of presence does not necessarily require the environment to be virtual, as can be experienced in a telephone conversation, where 'you are certainly aware of the person on the other end of the line (Copresence Illusion), and you can interact with that other person (Social Presence Illusion). However, you do not get the impression that you have been transported to another place.' (Skarbez et al., 2017, p. 5) Both definitions lend themselves to describe this study's use-case as it attempts to establish a virtual presence in the local physical space using a sonic avatar and provides a mediated form of interaction through verbal communication to develop an illusion of social presence.

As the notion of immersion also rather refers to a framework within a virtual environment, a conceptual definition is required for the relation between the user and the application environment constructed in this use case. Again, the survey presents two concepts of interest for this study. The terms '*Involvement and Engagement*' (Skarbez et al., 2017, p. 8) are introduced as mainly relating to the same concept in games and virtual environments alike (Skarbez et al., 2017, p. 8). They are roughly defined as 'a state of

focused attention or interest' (Skarbez et al., 2017, p. 8), and, citing Witmer and Singer, 'a psychological state experienced as a consequence of focusing one's energy and attention on a coherent set of stimuli or meaningfully related activities and events.' (Skarbez et al., 2017, p. 8) A third related concept is that of *flow*, defined as, citing Csikszentmihalyi, 'an optimal state of concentration, "the state in which individuals are so involved in an activity that nothing else seems to matter".' (Skarbez et al., 2017, p. 9) Furthermore, they cite Brockmeyer et al. in 'argu[ing] that flow, since it involves experiencing an altered state, may be a deeper state of engagement with media than presence' (Skarbez et al., 2017, p. 9). These alternative concepts to the idea of immersion shaping the experience within a virtual or mediated environment are a guiding conceptual basis for the design processes of the modes of expression and interaction within the telepresence application, as a deep involvement or flow would be beneficial to shape a feeling of 'togetherness' in a shared task of creating sound or music.

2.2. Motion capture

The positional tracking of specific key points on a moving body over time is commonly referred to as *motion capture*. The technique is often used in **CGI!** (**CGI!**), enabling puppeteering of 3-dimensional (3D) avatars for motion picture productions and character animation in games. High accuracy is required for these purposes, and the technological and financial entry barriers are relatively high. These applications use systems by Vicon[1] or OptiTrack[2], which work with visual markers to track movement in space and require a studio environment to be deployed. An example of a markerless optical system is Captury Live[3], which tracks humanoid moving actors with a 360° camera setup. In the performance field, the preferred methods are inertial measurement unit (IMU)-based

[1] https://www.vicon.com
[2] https://www.optitrack.com
[3] https://captury.com

tracking systems like the SmartSuit by Rokoko[4] or the Perception Neuron[5] sensor kit since they are independent of the lighting conditions. Both visual and inertial methods are available in variants from different manufacturers on a cost spectrum that varies from the low thousands to hundreds of thousands of euros in investment.

The 'grassroots' setup for motion capture is the Kinect, introduced by Microsoft in 2010, featuring an infrared time-of-flight measurement system that produces a depth image from which a pose can then be extracted using 3D pose estimation (see Ye et al., 2011). The Kinect was frequently used among creative coders, although it was initially developed for games. In 2023, Microsoft announced that the Kinect, now called Azure Kinect, would cease production, and its software development kit (SDK) would be handed over to Orbbec, another manufacturer of 3D-cameras (Mehta, 2023). Other low-cost 3D cameras are on the market, like the Oak-D[6] cameras with an integrated processing engine or the Orbbec Femto Bolt[7] supported by Microsoft. These systems produce relatively low accuracy but can be used as multi-camera setups or to analyse more general dynamics in the movement data.

Deep learning models for motion capture like PoseNet (Kendall et al., 2016) or BlazePose (Bazarevsky et al., 2020) have also become available and, while primarily used on 2-dimensional (2D) (surveillance) footage, can be extended into 3D if combined with the proper calibration data (e.g. depth images). These models are fast and can be run on a regular webcam. However, they also tend to produce relatively coarse movement data and do not generate reliable depth information when used on 2D information only.

[4] https://www.rokoko.com
[5] https://neuronmocap.com
[6] https://shop.luxonis.com/collections/oak-cameras-1
[7] https://www.orbbec.com/products/tof-camera/femto-bolt/

2.3. Movement data sonification

The term 'sonification' primarily refers to the auditory expression of data. Some well-known examples include the beeping sounds used for car parking assistance, the electrocardiogram machines used in hospitals to relay the heart rate acoustically, or the Geiger counter, sonifying ionisation to indicate the level of radioactivity in the environment. While the general method of expressing quantities acoustically can be traced back as far as 3500 BCE (Worrall, 2018, p. 178), the method of 'parameter mapping sonification' (Hermann et al., 2011, Chapter 15) is much more recent and is most commonly used today. Its emergence was predated during the eighteenth and nineteenth centuries by a shift in Western music towards more abstract and gesturally focused expression and formalised rules. It came into its current form through the emergence of serial music in the twentieth century (Worrall, 2018, pp. 179–180) and algorithmic composition as its descendant, popularised by composers such as Iannis Xenakis and John Cage, and with the emergence of electric and electronic means of sound generation.

The sonification of human movement data using parameter mapping is often used in health and therapeutic research to offer an acoustic interface to experience dynamics in movement properties. Examples are movement perception in rehabilitation and active movement practice as part of learning exercises (see Brock et al., 2012). It requires specific data points to be tied to acoustic properties. This can be a direct value connection from one property to another (e.g. velocity to loudness, altitude to pitch). However, it can also be achieved using indirect logical constraints expressed in more complex algorithms (e.g. if multiple thresholds are crossed, a single signal is triggered).

Today, the many possibilities for real-time data analysis combined with digital sound synthesis enable a broad spectrum of practical and artistic applications of movement sonification. Combined with the various means of motion capture available today, it can be used on stage to generate a real-time soundtrack to the movement or offline for recording, analysis and composition of music in a reciprocal process between dancers

and composers. Here, the particular mode of artistic expression is left open, and the focus lies on providing a framework for extracting movement qualities, transmitting these qualities, and generating events based on simple rules. The forms of concrete artistic expression made possible by this functional infrastructure are beyond the scope of this study.

2.4. Embedded computing and open-source hardware

The concept of an embedded system is defined as 'a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, embedded systems are part of a larger system or product, ...' (Barr, 2015) While this definition applies to most contemporary electronics, it rose to broader awareness through its popularity in the do-it-yourself (DIY) electronics community. In 2003, Hernando Barragán, a student at the Interaction Design Institute Ivrea (IDII) in Italy, created the Wiring project as his Master's Thesis, aiming 'to make it easy for artists and designers to work with electronics, by abstracting away the often complicated details of electronics so they can focus on their own objectives.' (Barragán, 2022) The Wiring project, after successful use in the curriculum at IDII, went on to become the basis for the Arduino project, launched in 2005 by Massimo Banzi and David Mellis as a fork of Wiring and without Barragán's involvement (Barragán, 2022). The Arduino development board line and its software ecosystem became the most popular framework for experimenting with open-source hardware and microcontrollers outside of the field of electronic engineering. At the same time, there are other successful projects like Adafruit Industries, SparkFun, RaspberryPI and more.

2.5. Web standards

The idea behind web standards is to provide stable definitions of core technologies that are used to build and present web content. Apart from providing a consistent display across different browsers, this is especially important for interacting with particular operating system (OS) or hardware functionality via the browser. As JavaScript (JS) does not define any specific input/output (I/O) functionality, it is the task of the browser environment to supply this. As the browser is the mediator between the OS and the web page, the idea of standardised application programming interface (API)s was devised and implemented. Several organisations standardise web technologies, with the most prominent of them being the W3C, Web Hypertext Application Technology Working Group (WHATWG), Ecma, Khronos and the Internet Engineering Task Force (IETF).

2.5.1. WebRTC

In 2010, Google acquired Global IP Solutions, a Swedish company developing real-time communication over internet protocol (TechCrunch, 2010). Their technology became the basis for WebRTC (Google, n.d.), which was subsequently proposed as a web standard and further developed by Google. It became an official standard in 2021 (Couriol, 2021), providing the functionality for transmitting video and audio streams over user datagram protocol (UDP) or transfer control protocol (TCP). Additionally, data streams with arbitrary message packets can be used to transmit binary or text data. WebRTC handles all low-level flow control and other transmission aspects and provides a simple high-level API. It can be used in direct peer-to-peer (P2P) setups where each party communicates with the others directly, a multipoint control unit (MCU) that receives all communication centrally and then broadcasts a composite signal to everyone, but also as a selective forwarding unit (SFU), relaying only the requested streams to participants and enabling one-to-many or many-to-many communication setups. The choice between the various setups has implications for scalability, infrastructure cost, privacy and security

aspects (Iyengar, 2021). Several software solutions for streaming media support the WebRTC standard, but in this case, focusing on the concept of a SFU is essential since it enables a more efficient load distribution, so the selection is reduced to the packages that support or explicitly focus on this type of topology.

Table 2.1.: WebRTC servers ranked by stars received on GitHub

WebRTC Server	Stars (k)
Janus Gateway, 2014	7.6
LiveKit, 2020	6.4
Mediasoup, 2014	5.7

The *Janus Gateway*[8] server is designed as a general-purpose solution, providing only the core WebRTC functionality and allowing developers to extend it using existing or custom-made plugins. This way, it can implement various schemes, such as P2P, MCU and SFU, but can also be used to create completely custom hybrids.

LiveKit[9] is a dedicated SFU server including SDKs for web, native mobile, desktop and server applications in various languages. It is developed and maintained by a relatively young company, as it was publicly released in 2021 and was 'started amid and in response to the pandemic' with the idea of providing 'free and open infrastructure capable of connecting anyone' (LiveKit, n.d.). While the software is free and open-source, a paid hosted service is also offered for those who want to experiment with real-time communication but want to avoid setting up an infrastructure. There are many examples of integration into existing frameworks, extensions for recording sessions on the server, as well as extended handling of streams.

Mediasoup[10] is different from the other options in that it does not present a standalone server architecture. It provides a versatile collection of Node, Rust and C++ libraries

[8] https://janus.conf.meetecho.com
[9] https://livekit.io
[10] https://mediasoup.org

that allow for building a custom server application from the ground up. While it takes care of the low-level RTC functionality, it provides somewhat granular building blocks to set up the actual implementation. This allows for building entirely decentralised peer-to-peer applications as well as server-centric setups. It was developed by a small team of contributors around its leading developers, Iñaki Baz Castillo and José Luis Millán.

2.5.2. WebSockets

The transmission protocol *WebSockets*, which was standardised as request for comments (RFC) 6455 by the IETF in 2011 (Fette & Melnikov, 2011), allows full-duplex communication between client and server, running on the same ports and transport layer seven as the half-duplex HyperText transmission protocol (HTTP) protocol, thus being compatible with existing web infrastructure. As the *WebSockets* standard was not fully supported across browsers for some time, there have been various approaches to providing real-time functionality to web applications more or less loosely based on the *WebSockets* specification. However, the current browser landscape shows much more complete support for the original *WebSockets* protocol (Deveria, 2024).

Table 2.2.: JavaScript WebSockets libraries

Framework	Stars on GitHub (k)
Socket.IO, 2010	59.7
ws, 2011	20.7
µWebSockets, 2016b	16.4

Socket.IO[11] is billed as a 'realtime [sic] application framework' (Socket.IO, 2010) and provides client and server implementation. While it is a popular choice for real-time communication in the browser, it implements its own protocol instead of building on the

[11] https://socket.io

WebSockets standard. As stated in the documentation, 'a WebSocket client will not be able to successfully connect to a *Socket.IO* server, and a *Socket.IO* client will not be able to connect to a WebSocket server' (Socket.IO, 2010).

WS[12] is a standards-compliant implementation of the WebSockets protocol for use in server-side applications written in Node.js. It is written in C++ to provide good performance, and it supports compression via implementation of the standards proposal in RFC RFC 7692 'Compression Extensions for WebSocket [sic]' (Yoshino, 2015).

µWebSockets[13] is focused on robustness and performance while exclusively communicating via standards-compliant WebSockets protocol. Like WS, it is implemented in C++, used via Node.js on the server side, and does not require a specific client library.

2.5.3. WebAudio

WebAudio, the standard for handling audio in the browser, takes care of basic mixing of channels and different sources (e.g. media streams, audio files). It can also be used for generating sound via synthesis nodes, and entirely custom audio nodes can also be developed. Another feature commonly used in games or virtual reality experiences is the possibility of placing sound sources on virtual soundstages rendered as ambisonics for psychoacoustics in headphones. Several frameworks provide a high-level abstraction to the WebAudio API and thus enable a speedier development process. While the selections of frameworks presented for selection are supposed to be the top three entries based on GitHub Stars, this list adds the relatively new framework *Elementary*, which takes a different approach to development using the declarative definition of sound structures. It has roughly half the rating of *Flocking* but has been around only since 2022 (2.3).

[12] https://github.com/websockets/ws
[13] https://github.com/uNetworking/uWebSockets

Table 2.3.: Popular JavaScript audio frameworks

Framework	Stars on GitHub (k)
Howler.js, 2013	22.6
Tone.js, 2014	12.9
Flocking, 2011	0.7
Elementary, 2023	0.3

Howler.js[14] is a complete audio framework that builds on the *WebAudio* API and provides easy access to audio functionality, focusing primarily on interactive audio for web applications or games. It offers various modes of sound playback, mixing, and spatial audio as a plugin. Still, at the time of writing, it only supports connecting live audio sources through a yet unmerged pull request on GitHub (rafern, 2022).

Tone.js[15] is explicitly focused on musical application, working much like a digital audio workstation software, providing various modes of sound synthesis, as well as transport controls, a meter and scales. It supports spatialisation using a 3D panner node and, while not explicitly documented, should support external audio stream input through its 'UserMedia' node.

Flocking[16] is more of an outsider, being around since 2011 but having gathered only a small amount of star ratings. It follows a different approach in that it defines sound objects using JavaScript Object Notation (JSON), making them portable and allowing for generative approaches to sound generation on a meta-level. According to its developer, 'its goal is to promote a uniquely community-minded approach to instrument design and composition.' (Flocking, 2011) Unfortunately, it currently does not support parallelising audio rendering in special workers. Thus, 'Flocking is not currently well-suited to applications that involve a lot of graphics rendering or user interaction.' (Flocking, 2011)

[14] https://howlerjs.com
[15] https://tonejs.github.io
[16] https://flockingjs.org

Another audio framework following a declarative and functional approach is *Elementary*[17], which has only been around since early 2023. It separates the declarative API for creating instruments and musical structures from the sound rendering, allowing it to use the Web Audio API for real-time audio in the browser and an offline renderer for Node.js. The framework offers extendability through native nodes developed in C++ and used in the Node.js environment.

Resonance[18] is a library worth mentioning, as it focuses exclusively on spatial audio. Google developed it based on Omnitone[19], another one of their projects focusing on ambisonic spatial audio rendering in the browser. Resonance received only one release and has remained dormant, but it still works without breaking changes. It uses a default head-related transfer function (HRTF) to model audio spatialisation and allows a virtual room to be created with different materials for walls, floors, and ceilings that provide different reflection types. Custom sound sources can be defined, connected to web audio nodes, and positioned around the virtual space. The library hooks into any existing audio context, thus allowing a combination with virtually any other WebAudio-compliant audio framework.

2.5.4. WebXR

The various virtual and augmented reality devices available are accessible via the web mixed reality (WebXR) API, with the browser bridging the communication with the headset and controllers. A 3D scene created in a web-based graphics framework like THREE.js or A-Frame can be instantly experienced on a virtual reality (VR) headset like the HTC Vive or Oculus Quest.

[17] https://www.elementary.audio
[18] https://resonance-audio.github.io/resonance-audio/
[19] https://github.com/GoogleChrome/omnitone

Table 2.4.: JavaScript 3D frameworks

Framework	Stars on GitHub (k)
three.js, 2010	97.1
Babylon.js, 2013	22
A-Frame, 2015	16

Three.js[20] is a 3D graphics framework with a large community of over 1800 contributors that has been around since 2010. It was initially developed for the ActionScript language used in Macromedia and later Adobe Flash (another Ecma standard-compliant language, see section 2.6). It features an extensive toolset for graphics generation, rendering and effects and relies on the WebGL standard to allow performant rendering via local graphics hardware.

Babylon.js[21] is another fully-fledged 3D framework with a strong focus on games and realistic, high-quality rendering, which was initially developed by Microsoft employees in 2013. The framework features an extensive collection of tools for interaction and animation and supports integrating the WebXR standard to use VR equipment in the browser.

A-frame[22] is a framework that allows the developer to create 3D scenes by composing custom HyperText markup language (HTML) elements that provide geometric primitives, lights, cameras, etc. This way, it has a comparably low entry barrier for people with limited scripting experience. It explicitly focuses on mixed reality applications, implements the WebXR standard and has preset control objects for various headsets and controllers. It is based on Three.js, whose full low-level functionality can be accessed through A-Frame if a more complex functionality or custom behaviour is required.

[20] https://threejs.org
[21] https://www.babylonjs.com
[22] https://aframe.io

2.5.5. WebBluetooth

This relatively simple API provides access to the computer's Bluetooth functionality. It allows connecting custom Bluetooth senders like Arduinos or other embedded devices with sensors or other DIY electronics, sending and receiving messages to and from the browser (see: section 2.4).

2.6. Programming languages

Table 2.5.: Ranking among the most used languages on GitHub (Daigle & GitHub, 2023)

Language	Rank
JavaScript	1
Python	2
TypeScript	3
C++	6

JS is a scripting language created by Brendan Eich in 1995 as part of the release of the Netscape 2 browser (Netscape, 1995), then officially standardised by the Swiss standards body Ecma in 1997 as ECMA-262 or ECMAScript, as it is known today. This standard became the basis for JScript by Microsoft and ActionScript as part of Macromedia Flash. The version currently being supported by all browsers (except Internet Explorer 11) is ECMAScript 6 (ES6) (W3Schools, n.d.). JS is an object-oriented, weakly-typed programming language that allows multiple programming paradigms. It is primarily used in the browser to add extra functionality to web pages. The underlying ECMAScript standard does not define any input or output methods, which means that this functionality should be provided by the specific environment it is being used in (e.g. desktop or mobile browsers), bridged by the web standards APIs.

TypeScript (TS) was released by Microsoft in 2012 'to accommodate an increasing number of developers who are interested in using JavaScript to build large-scale Web applications to run in a browser rather than on the desktop.' (Jackson, 2012) It complies with the underlying Ecma scripting standard and is designed as a superset of JS, adding static typing. As it has to produce regular JS code to be executed in the browser, it uses a compiler and also allows for mixed scenarios in which TS is used alongside JS.

Node.js was initially released by developer Ryan Dahl in 2009 as a server-side JS environment. Node.js runs standard ECMAScript in Google's V8 engine, allowing multithreading and native code integration. It was initially sponsored by the company Joyent but is now maintained by the OpenJS Foundation after some disputes about stewardship in the developer community. Node uses the Node Package Manager (NPM) to package code as modules, which can be used as dependencies, which has become the de facto standard for JS dependency management both for Node.js and browser-based applications. The modules can also integrate native C++ code, enabling bindings to most open-source libraries in the Linux ecosystem. It can be used to develop APIs or other server-side applications and supports local web development processes like preprocessing, packaging, and deployment.

The *Python* programming language was created by Guido van Rossum in 1990 (Python Software Foundation, n.d.). It is a multi-paradigm language that is both dynamically and strongly typed (van Rossum, 2008). It relies heavily on indentation and whitespace to structure the code. The language uses a standard library, and the surrounding ecosystem of available modules and applications based on Python makes it a popular choice for data processing and science. There is a native code interface that allows extending Python with bindings to native code, similar to Node.js.

C++ originated as an extension to C in 1985. It is a multi-paradigm, statically typed and object-oriented programming language. It is used to develop code for embedded open-source hardware platforms (see: section 2.4), extend both Node and Python,

and, more generally, provide direct interaction with the operating system and its APIs. While it is the oldest of the programming languages mentioned here, it has remained essential to the open-source world, not least because of its prevalence in the Linux ecosystem.

2.7. Application design paradigms

The idea of the *single-page application (SPA)* originated around the beginning of the 2000s with the concepts 'Inner-Browsing' (Galli et al., 2003) and asynchronous JavaScript and XML (AJAX) (Garrett, 2005). It breaks with the traditional way of moving from one page to another in favour of asynchronous loading and replacing parts of the current web page. This allows a website to evoke the look and feel of desktop applications by not reloading the entire page for updates.

The term progressive web application (PWA) was initially coined in 2015 by two Google employees in an online Article (Russell & Berriman, 2015). At its core, it describes the process of a website 'progressively' evolving into a fully-fledged on-device application by adding offline functionality and blending with the operating system. It is often built atop the concept of an SPA and can be perceived by the user as an application they installed locally instead of accessing it at a remote location.

A *real-time web application* enhances the user experience by relaying relevant changes on the server to the client as they happen. This can be a simple chat application or a more complex collaborative multi-user environment. While real-time updates can happen on any multi-page website, they can also be a beneficial feature of an SPA or a PWA. Instantaneous updates are commonly realised using WebSockets, allowing updates to be pushed to the client whenever a resource on the server changes.

2.8. Frontend frameworks

There is a broad range of available JS frameworks to build dynamic frontends for SPAs and PWAs. The three libraries currently dominating the landscape are React, developed by Facebook in 2013, and Vue, created by Evan You in 2014. These libraries can be used with frameworks to offer complete routing and state management solutions. Another popular framework is Angular, initially released by Google in 2010 and re-released in 2016.

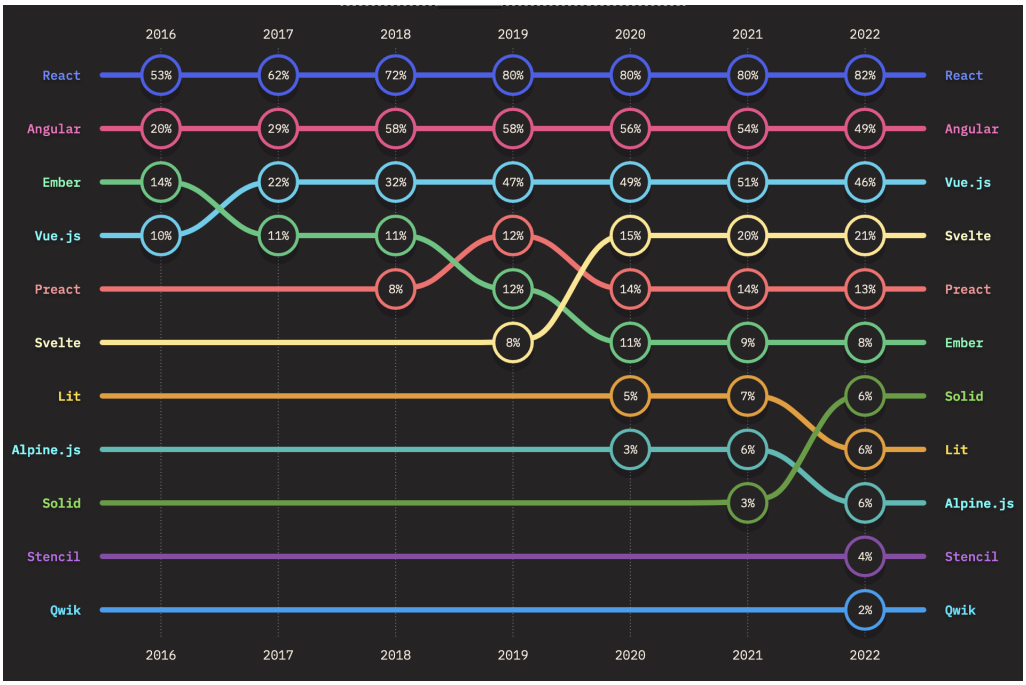

Figure 2.1.: State of JS: Most used frontend frameworks in 2022 (Greif & Burel, 2023c)

React[23], developed by Facebook and maintained by its successor Meta, has become the most widely used tool for building SPAs. It is steadily leading the rankings for most used frontend frameworks both in the Stack Overflow (Stack Overflow, 2023b) and the State Of JS (Greif & Burel, 2023c) polls. By definition, it is not a framework but a user interface (UI) library that relies on other extensions to support state management, routing and deployment functionality. Although it is not a framework itself, there are

[23] https://react.dev

existing frameworks like Next.js[24] for the web and ReactNative[25] for building mobile apps using native functionality. React makes use of JavaScript XML (JSX), which allows directly mixing inline HTML with the JS or TS source code.

Vue[26] was developed by Evan You and is maintained by an international team of individuals. It had a relatively marginal presence in the US and Europe in the first years after its inception. This can be partially attributed to its origin in China, as most of its supporting modules were localised in Chinese. Over the years, it grew in popularity and received more international support, eventually overcoming the language barrier. Unlike React, it is billed as a 'progressive framework' that provides fundamental functionality for building reactive components but also accommodates more complex use-cases (You, 2021). Vue builds on standard JS or TS, HTML and Cascading Style Sheets (CSS) to create components, recommending a simple template mechanism mixed with reactive substitutions. However, it also supports using JSX for specifying inline HTML within JS. As with React, there are extensions and frameworks like Quasar[27] and Nuxt[28] that enable more sophisticated workflows for application development and deployment.

Angular[29] was initially released by Google in 2010 as AngularJS and officially discontinued in 2022. A completely overhauled version 2 was released in 2016 and is maintained by Google. It differs from React and Vue in that it is a complete framework containing everything required to build and deploy an application, and it explicitly recommends TS as a programming language. The framework is also less flexible because it is more opinionated and has its own set of best practices baked into the framework's structure.

[24] https://nextjs.org/
[25] https://reactnative.dev/
[26] https://vuejs.org/
[27] https://quasar.dev
[28] https://nuxt.com
[29] https://angular.io/

2.9. Backend frameworks

The *Express*[30] framework provides the basic functionality to create web servers, including routing and middleware functionality. TJ Holowaychuk developed it in 2010 and sold it to StrongLoop (Tsang, 2014), which IBM subsequently acquired (Yegulalp, 2015). It is currently under the stewardship of the OpenJS Foundation (Krill, 2016). Express has become the de facto standard for building web services in JS, leading the ranking in the State of JS survey (Greif & Burel, 2023b). Although it contains the necessary parts to create a web service, it does not enforce a specific architecture, which can be problematic for maintaining a robust application structure. For developers who prefer a more explicit structure, various other frameworks that add more opinionated structures or extensions are built on top of it.

Billed as a successor to Express, *Koa*[31] is developed by the team behind Express. It aims to provide a more robust and minimalistic iteration of the middleware-based architecture of Express. Like Express, it allows for building a service from scratch in free form but is also the basis for other, more explicitly structured frameworks.

Other frameworks and a more stringent and structured application structure might be more desirable for complex applications. Numerous JS frameworks, some based on Express or Koa, and others that provide their own basis for routing. To review all possible options is beyond the scope of this study. In the following, three frameworks are selected for their specific nature related to popularity and stability, with an explicit focus on real-time applications.

Nest[32] is a backend framework for developers looking for a more strictly opinionated and robust setup than Express, e.g. for enterprise applications. It follows a modular concept, making dependencies available to the services via injection. Multiple database

[30] https://expressjs.com
[31] https://koajs.com
[32] https://nestjs.com

Table 2.6.: State of JS survey: Most used backend frameworks (Greif & Burel, 2023b)

Framework	% of question respondents	Stars on GitHub (k)
Nest, 2017	30.2	62.7
Feathers, 2011	8.8	14.9
Meteor, 2012	2.7	44

options exist, and transports can be both HTTP and WebSockets. There are command-line interface (CLI) scripts that enable automatic generation of boilerplate application code, and the language used to build Nest applications is TypeScript. It ranks second among the most-used backend frameworks in the State of JS survey (Greif & Burel, 2023b).

The *Feathers*[33] framework takes a different approach, making few assumptions about the specific application structure. It uses aspect-oriented programming, a service-centric architecture and before-, after- and around-hooks (so-called 'cross-cutting concerns') for the services that modify basic behaviour or add functionality. There are adapters for a wide range of databases and authentication methods. The framework has a dedicated concept of channels that enable real-time functionality and messaging to clients. Real-time transports are abstracted and can be deployed using Socket.IO or standards-compliant µWebSockets (see section 2.5). It also provides a CLI to generate application code in JS or TS. Feathers started as a hobby project by David Luecke and Eric Kryski in 2013 (Kryski, 2016) and is currently maintained by David Luecke and a community of individual contributors. It still ranks in the lower percentages in the State of JS survey (Greif & Burel, 2023b) but almost doubled that percentage from the previous one in 2021 (Greif & Burel, 2022).

Meteor[34] focuses explicitly on real-time applications using WebSockets. The framework is an outlier because while its core is open-source, other parts are proprietary code.

[33] https://feathersjs.com
[34] https://www.meteor.com

Nonetheless, it should be mentioned because it has been around for over ten years and uses WebSockets exclusively. It was released in 2012 by a startup company, immediately received venture capital funding from Andreessen Horowitz and was eventually sold to Tiny Capital in 2019 (Lardinois, 2019). The framework primarily uses MongoDB as a database system and initially provided its own package manager and ecosystem, build system, and template system based on Mustache. Meanwhile, this exclusive strategy has been abandoned in favour of adopting the Node Package Manager. Still, it seems to be subject to debate regarding its ease of use versus its 'growing pains' and related trouble with wide adoption (doppp & forum users, 2019).

2.10. Databases

Table 2.7.: Stack Overflow Developer Survey 23: The top three multi-user databases (Stack Overflow, 2023a)

Database	% of all question respondents	Stars on GitHub (k)
PostgreSQL, 2010	45.55	14
MySQL Server, 2014	41.09	9.9
MongoDB, 2009	25.52	25

PostgreSQL[35] is a very widely used database which uses a table-based data topology and implements structured query language (SQL) for interaction with the database and its contents. The relatively rigid database schema provides a solid structure for data storage and retrieval but, on the other hand, requires migrations to be written to transition from one database structure version to another. It has an extensive feature set supporting complex data structures, geographic information system (GIS) data and data structured in JSON format. Developed in the 1980s at the University of California and switched to the SQL in the 90s, it has remained a popular choice for enterprise and small-scale use.

[35] https://www.postgresql.org

Similar to PostgreSQL in that it also uses SQL, *MySQL*[36] supports many of the features of PostgreSQL, but has an overall smaller feature set. It was initially developed in the 1990s by the private Swedish company MySQL AB and was forked as a completely open-source version in 2009 and renamed MariaDB[37]. It is still a popular choice, especially for smaller web projects that don't need the extra functionality and value its relatively simple setup.

MongoDB[38] is a document store database that is designed to hold large amounts of unstructured data. It has its own query language and features aggregation functionality that allows map/reduce and transformation operations or resolving of relations on the data before being sent to the client. Although it uses the Not-only SQL (NoSQL) paradigm and allows storing documents of any kind in a collection, it eventually added the option of using schemas for validation. It was initially released as open-source in 2009, then was put under a proprietary license in 2018, but remains available to be used for free with limited support.

2.11. Application deployment

Containerisation, in the context of computing infrastructure, refers to the 'packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure.' (IBM, n.d.-b) It was popularised through the release of the Docker Engine[39], an open-source project devoted to creating an industry standard for application containerisation (Barbier, 2014). The Docker team eventually launched the Open Container Initiative (OCI) in 2015, which serves as 'a lightweight, open governance structure (project), formed under the auspices of the Linux Foundation, for the express

[36] https://www.mysql.com
[37] https://mariadb.org
[38] https://www.mongodb.com
[39] https://www.docker.com

purpose of creating open industry standards around container formats and runtimes.' It subsequently received Docker's container runtime and format as a donation, which was released as runC version 1.0 in 2020 (Linux Foundation, n.d.). It has recently become the de facto standard for packaging and delivering applications in web development and beyond. GitHub reports that 'in 2023, 4.3 million public and private repositories used Dockerfiles — and more than 1 million public repositories used Dockerfiles for creating containers.' (Daigle and GitHub, 2023)

Container orchestration builds on the concept of containerisation. 'Container orchestration automates the provisioning, deployment, networking, scaling, availability, and lifecycle management of containers.' (IBM, n.d.-a) The concept first gained popularity as Docker 'swarm mode', a functionality of the Docker software. Still, its most successful instance so far is the software package Kubernetes[40], which originated at Google in late 2013 (Burns, 2018) and went on to be included in the Cloud Native Computing Foundation (CNCF), a project by the Linux Foundation, that 'aims to advance the state-of-the-art for building cloud-native applications and services' (Linux Foundation, 2015). It can be extended, highly customised and deployed on anything from an embedded device to a large-scale cloud infrastructure, providing a versatile deployment and management tool for many application infrastructures.

[40] https://kubernetes.io

3. Methodology

This feasibility study was based on two essential parts. The first was a reference implementation, providing insights into the work necessary for a basic functional implementation. Based on this implementation, an analysis of the application's functionality was made, an overview of the time spent on development, a qualitative review of the resulting codebase, and a reflection on the work process.

3.1. Reference implementation

To produce a valid test subject for the proposal, the reference implementation was created according to a prior selection of tools and methods deemed appropriate for the project. The choices were made from the concepts and tools presented in chapter 2.

First, possible candidates were identified through internet research, and then at least three candidates were selected using the number of 'Stars' received on GitHub as an indicator of popularity. In some cases, other data sources had to be used where the technology predates GitHub (e.g. databases or programming languages), and its popularity should be judged by other means. Developer surveys are being conducted by the popular technology forum Stack Overflow with over 90.000 participants for 2023 (Stack Overflow, 2023c) and the 'State Of JS' survey with over 20.000 participants that is more focused on web development (Greif & Burel, 2023a). Additionally, GitHub publishes a yearly statistic on its public repositories, which helps identify technological trends and popularity among millions of open-source repositories (Daigle & GitHub, 2023). These sources provided

additional input on popularity among a specific professional field and a general overview of technologies actively used. The decision on choosing a candidate was made not by popularity alone, but with a stronger weight on a good fit to the project's specific requirements and needs. If a less popular framework did fit the specific development style, it would still be preferred to the popular status quo. Another case might be a more recent project that has yet to collect as high of a rating on GitHub but presents a promising new approach or feature set.

The application development worked from the most basic boilerplate code towards finding the appropriate structure for the specific use case. Well-known and easily defined components were built first. The special functionality was then built on top in constant cycles of adding functionality, reviewing the codebase and refactoring towards abstraction and separation of basics from specifics. As only a rough architectural model for the project was defined beforehand, tests and documentation were written later in the process, as the parts stabilised on their own and in their relationship with each other. This method does not strictly adhere to common development procedures but borrows loosely from agile development (sprints, reviews, adjustment) and simple forms of the ideas put forward in the book ‘Pattern-Oriented Software Architecture’, such as application partitioning through layering, separation, and standardised messaging (Buschmann et al., 2007). A more systemic approach for the development process, like application modelling using unified modeling language (UML) and test-driven development, might be desirable for teams, but the various and disparate ‘moving parts’ in conjunction with heavy reliance on browser-only APIs complicate the creation of a well-simulated testing environment using either real or mock-data, especially for a single developer.

The application was implemented in its entirety, documented and packaged. Appropriate test coverage was provided for the core functionality in the API and the messaging components, and the overall time spent was logged in timesheets and categorised by the general work areas. The application's server components were deployed early on to university hardware and made available over the internet. The client application

was then run and tested on exemplary computer systems in and outside the university network.

3.2. Analysis and evaluation

A statistical analysis of the timesheets provided insight into the time spent on various aspects of the software. It differentiated between basic boilerplate code that can be reused and custom code used for the actual use case to provide insights both into the feasibility of setting up such a system from scratch with only a single developer, as well as the potential cost of just reworking the parts deemed transient and related to the specific use case.

The application's performance was tested regarding the load put on the central processing unit (CPU) (server and client) and the network throughput and latency. It is verified that all message processing works as expected through unit testing and simple testing tasks performed on the application. A practical test that analyses the actual user experience and uses performers and real dance interaction is beyond the scope of this study.

The *code quality* was assessed based on volume (lines of code without comments) and cognitive complexity, (see Campbell, 2023), which is an updated version of the older concept of cyclomatic complexity, which counts the number of linearly independent paths throughout a piece of code (see McCabe, 1976). It was published by the Swiss company SonarSource[1] and 'has been formulated to address modern language structures, and to produce values that are meaningful at the class and application levels.' (Campbell, 2023, p. 4) It accounts for modern control structures aims to facilitate legibility by rewarding a code style that benefits general understandability (Campbell, 2023, p. 4). Both metrics should be kept low for each source code file, as code with many lines makes it harder to read, and high complexity is more challenging to follow and understand. The code volume

[1] https://www.sonarsource.com/

is difficult to reduce to a static maximum, but ‘the [Software Assurance Technology Center (SATC)] has found the most effective evaluation is a combination of size and complexity’ (Rosenberg et al., 1998, p. 6), so a self-imposed threshold should be set, even if exceptions are made later. For object-oriented programming (OOP), limiting a file to a single class is often advised. Still, depending on the language used, the file can also contain multiple functions that are better grouped instead of scattered across various files. A general rule of thumb that has proven to be a good measurement from past experience is that a single file should not exceed 160-200 lines of code (without comments) to skim it and get a general overview quickly. There are exceptions to these rules where a specific function or class exceeds this threshold, but breaking it up would not make it easier to understand. In those cases, there can be a discussion about changing the general application design to partition the functionality differently. Alternatively, it can be decided to keep this code and improve documentation. Reducing volume and complexity is especially important if the code should be passed on to others who want to maintain and modify it for further use. Here, the general idea is that the transient (hackable) parts (e.g. UI) should be as simple as possible, avoiding unnecessary complexity, and the more static and stable core parts can be the location where the more complex parts are moved to (e.g. the core SDK).

Additionally, a *critical reflection* and analysis of the development process was created to weigh the expectations against the experiences while implementing the decisions made in planning the application. It should evaluate general reproducibility and feasibility and discuss the benefits and drawbacks of establishing a task-specific application from scratch.

4. Application concept

The reference implementation was named *Sensorama*, a reference to the early immersive experience patented by filmmaker Morton Heilig in 1962 (Heilig, 1962). It is said to be the first virtual reality systems (5 Gigante, 1993) and features stereoscopic video and audio, simulated wind and even olfactory stimuli. The name is a nod to the longstanding history of ideas around immersive experiences mediated through machinic technology and the multisensory approach that, in this case, refers to the possible input and output methods. While the application's base functionality could theoretically be used for any number of participants, only limited by the infrastructure's resources, it was specifically designed for only two active participants at a time. This decision was made because it would already be a challenge to adapt to relating to a mediated presence only by localising it acoustically and at the same time focusing on sound cues to base one's own movement on. However, a passive option of viewing the virtual space was included, allowing the use of WebXR functionality to join as a spectator.

4.1. Architecture

Due to the containerised packaging and deployment, the application can be deployed in any cloud environment or other hosting platform providing virtual or dedicated hosts with root access. The underlying software infrastructure requirement is minimal, and the required components are a Linux OS with installations of Docker (with ContainerD) and Kubernetes. No special hardware is needed, and the system can run in any environment

that provides network access, storage space and standard computing resources. The underlying general architecture was partitioned into three basic parts (Figure 4.1). The general infrastructure is thought of as any kind of Kubernetes installation, either on bare-metal or a virtual server instance. Running on top of that is a set of base applications (API and WebRTC server), which would ideally be the same across multiple deployments of the actual implementation. The UI, which holds the use-case specific fucntionality, could then be forked, modified and duplicated multiple times, but using the same basic architecture.

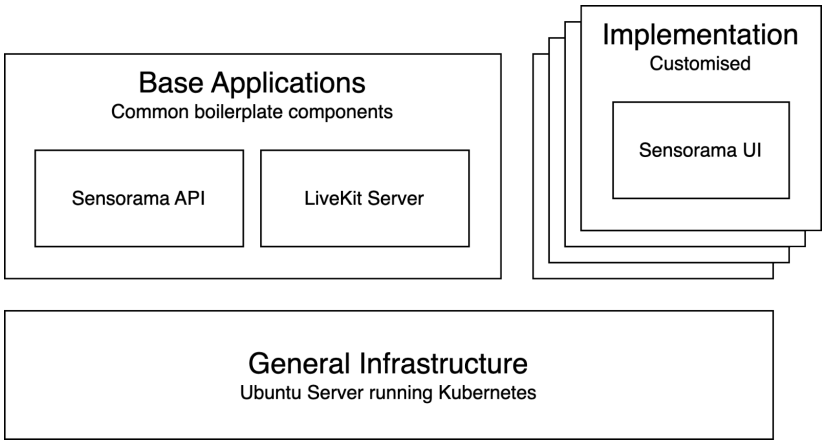

Figure 4.1.: Partitioning the Sensorama technology stack

While all the frameworks presented in Figure 2.1 could be used to build an application as envisioned in this study, Vue was selected due to the relatively high acceptance and the comparably easy learning curve. While it might not be the choice for large-scale or enterprise apps, the low entry barrier and the simple structure make it ideal to get an app up and running quickly, experiment with it and pass it on to others for hacking and custom modifications. The Quasar framework was used to accelerate and simplify the initial development. It extends Vue's basic functionality with a UI library complete with layout tools, common interface elements and a comfortable development and deployment environment.

The choice for a backend framework landed on Feathers and, by extension, Koa. The simple structure and code generators allow for a speedy setup and deployment of a

simple WebSockets API that provides authentication and resource management. It was connected to a MongoDB database because there was no definitive initial plan for how the stored and retrieved resources would be structured and typed. Using a flexible document store, the data could be easily overwritten with updates and then wiped before the schema would eventually be deemed stable.

LiveKit was chosen as the WebRTC server implementation because it is versatile, and there is an easy installation method to set it up as a container running alongside the Redis database in Kubernetes. While Mediasoup might have allowed a more precise implementation and probably a more efficient one, the workload overhead for building everything already offered by LiveKit was deemed too much effort for this kind of application.

4.2. Design paradigms

The basic design paradigm chosen for the Sensorama application was that of an SPA. As a remote API was already involved in managing access to shared resources, the extended PWA paradigm was not helpful for this scenario. It was designed as an exclusively real-time application that uses WebSockets for all transmission between app components and uses the WebRTC standard for user communication. The project was structured as a monorepo, where all components are developed across different languages in a single repository.

The application's custom part was partitioned into the user interface, which was deployed as a static built HTML/CSS/JS bundle, the API, which was set up as a single-process Node.js application and the so-called ‘Data-Producers’, which are external native utilities written in Python and C++ that provide bridges to motion capture hardware.

The primarily favoured coding paradigm was OOP, but this was not strictly enforced for all components. As some frameworks prefer different, more functional paradigms that are also compositional (Vue) or aspect-oriented (Feathers), it was deemed beneficial to refrain from enforcing a singular coding style. While this might usually be considered bad practice in terms of maintainability for long-term development, it served the purpose of a modular and somewhat transient 'single-use' application structure.

An essential part of the development concept was the sequence of development phases. As there was no explicit definition beforehand, the development started by establishing a functional skeleton first and worked within that to carve out the actual functionality. Initially, monolithic large blocks of code were built. Various approaches to desired functionality were quickly tried and discarded or kept and subsequently extracted into separate components grouped by functional association. Using this strategy, it was essential to review and refactor regularly and often to solidify the application structure and prevent it from dissolving into 'spaghetti code' and to avoid unnecessary side-effects among the components. The core features were extracted into a separate SDK module towards the end of the development process, and appropriate unit tests were written. As the application could now move into practical testing and, thus, some sort of 'production' deployment, the core functionality needed to become more rigid and covered by test cases.

The general user interface and data producers were considered transient because they served only the singular use case and should be subject to frequent future modifications. These application components should be hackable and replaceable, so they were not formally tested, at least in the scope of this study. The unit testing focused on the data input and output for the core functionality to provide a stable foundation by keeping all components connected in a unified messaging system. By modelling the basic request and response cases and formulating them as tests, potential later users would also have a tangible way of understanding the application's core mechanics.

For JS, three popular testing frameworks are Jest, Mocha, and Jasmine (4.1), among others, that can be used for implementing unit testing for the project. In this case, the selection skipped the most popular option of Jest in favour of Mocha, which is used by the Feathers API framework in its generator for boilerplate code. This way, basic tests to base work on were already available and, to keep the project consistent across modules were adopted for the core SDK module as well.

Table 4.1.: Popular JavaScript testing frameworks

Framework	Stars on GitHub (k)
Jest, 2013	43.2
Mocha, 2011	22.4
Jasmine, 2008	15.7

4.3. Movement quality extraction

One quality chosen was the ‘average velocity’, generally defined as the distance travelled over time (e.g. m/s). Additionally, two more specialised concepts, developed for expressive movement analysis, were selected: ‘Quantity of Movement’ (Volpe, 2003, pp. 96–97), describing the amount of difference between poses over time, and the ‘Contraction Index’ (Volpe, 2003, p. 97) which observes the density of space used by a pose. While the former is easily defined as the velocity calculated for a singular point (the centre of mass with an equal distribution), the latter two have been initially developed to observe pixelated 2D camera images and needed to be translated to 3D space.

4.4. Sonification method and sound spatialisation

The three basic movement qualities extracted provided the basis for a reference pipeline implementing parameter mapping sonification (see section 2.3). The movement qualities were made available to the application, enabling an event-based system where thresholds that switch a boolean value at the crossing time can be defined. As these values are watched, events can be triggered depending on their on- or off-state. At the time of an event, the current value of a quality or a combination of them can be used to determine the magnitude of the event's impact.

A simple system of scales, chords, and note selection was introduced to create an example sonification algorithm. The system was based on selecting a specific scale (e.g. E-minor) for which a list of possible chords could then be produced. Each time an event is triggered on one of the qualities (e.g. Quantity of Movement), the actual value can be used to decide which chord to select from the list using the normalised value. The notes selected from the chord are then transposed using another quality (Contraction Index), selecting the root note for the chord across the octaves. The average velocity was used to set the note length, triggering shorter notes when standing relatively still and more extended notes while travelling in space. It must be noted that while this is a valid example for a parameter mapping sonification and will produce largely harmonic notes, it falls short of producing an output of a more complex musical quality because it lacks the ability to construct deeper structures and dramaturgic variance. However, if engaged over a more extended period and practised, it could still yield interesting results, but this would also require a very specific way of composing the movement required to produce musically exciting results.

Sound spatialisation was achieved by playing the notes through a panner node provided by the audio framework and the WebAudio API. Both the audio stream retrieved from the WebRTC connection was connected to a panner node, and the sound produced from the synthesiser nodes for the local and remote participants. If the participant's head

orientation and position were available through motion capture or the head tracking device, this spatial information would also be applied to the listener node and the other spatial audio nodes.

4.5. Messaging

Standards-compliant protocols enabled streamlined messaging among the disparate application components based on different languages and run in various environments (see Figure 4.2). Starting on the client side, the motion capture data producer starts a local WebSockets server to which the web application running in the browser can connect and receive live data. The browser application can connect to the custom head-tracking device using the WebBluetooth standard and receive data messages using the Generic Attribute Profile (GATT)[1].

The conferencing functionality implemented in the web application was used to send local microphone audio and to relay the local producer utilities'data to other participants via the LiveKit server using the WebRTC protocol. In the backend, the LiveKit server can push status updates as HTTP webhook calls to the API server to notify it about connects and disconnects. The API server uses the WebSockets protocol to relay updates on persisted data and LiveKit update events to the client browser and receives authentication and general data requests.

[1] https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-gap-gatt/

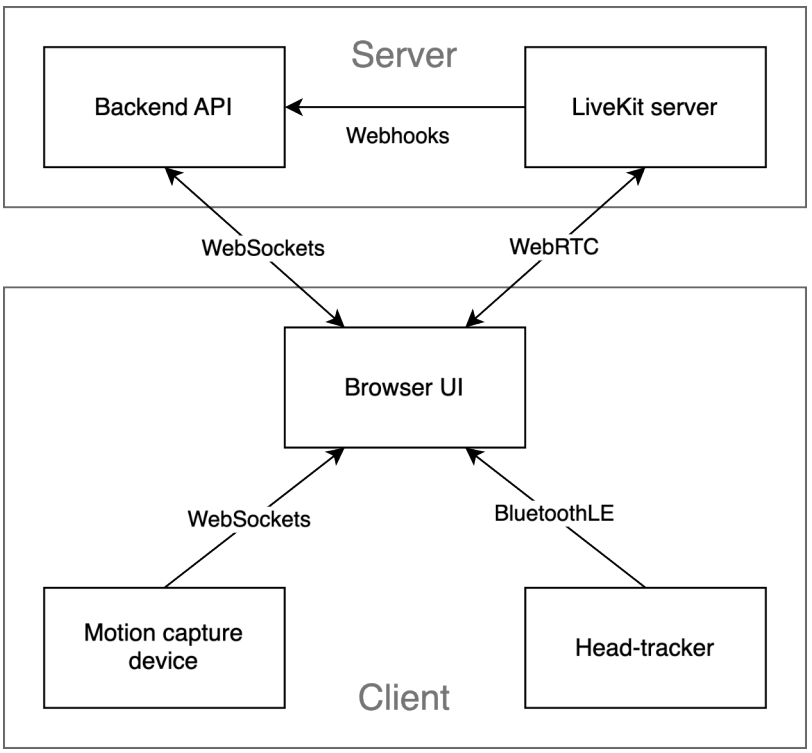

Figure 4.2.: Messaging flow between the application's components

4.6. Data modeling

Four core data models were defined for the application (see Figure 4.3). The two models persisted by the API server are *Spaces* and *Users*. These were modelled as simple reference objects providing the basis for connecting users to virtual spaces, akin to spatial chat rooms, and each User can own multiple Space objects. A Space is a container object representing a shared space populated with multiple participants'sensor readings. Users can request one or more *Token* objects that allow them to connect to a 'room' on the LiveKit server that maps to a specific ID of a Space. Once connected, the LiveKit server notifies the API server of the new connection, and now the connected User's ID can be found in the list retrieved from the virtual 'connected' property of the Space object.

The connected participants can exchange arbitrary *data messages*. The data messages were not designed to be encoded as JSON text messages but sent as raw data to make

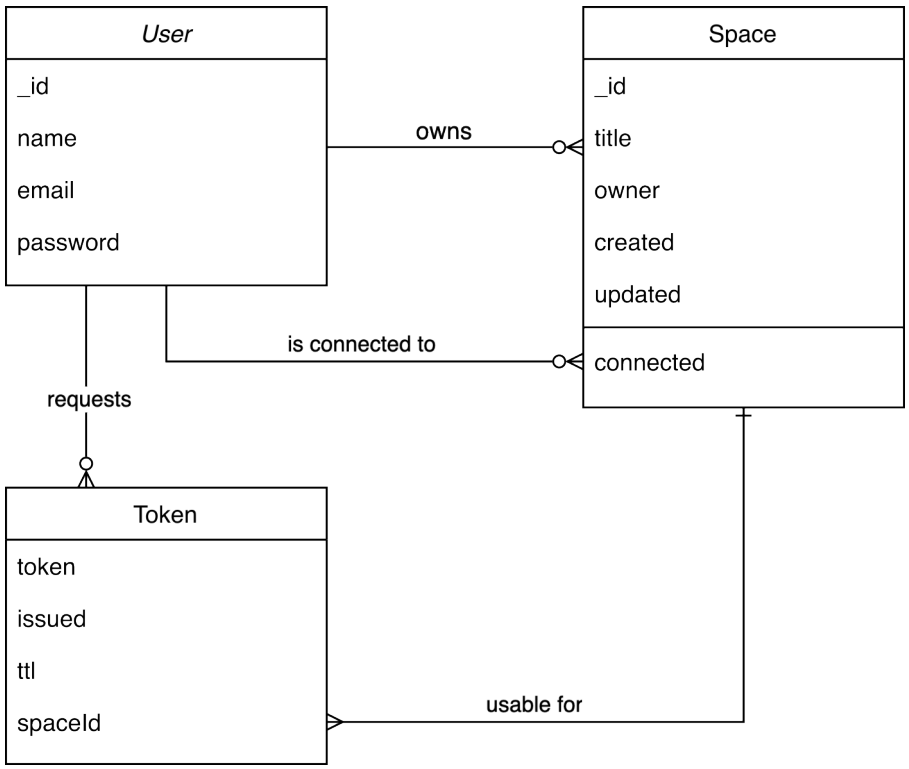

Figure 4.3.: Basic data model used in API server

them as small as possible. Messages were structured as byte sequences, with a 64bit long integer timestamp using the first eight bytes, then a single byte with an unsigned integer for selecting a message schema from the enumerated message types and a freely defined sequence of different number types (Figure 4.4). 32-bit floating point numbers were used for all sensor readings as the numbers stay sufficiently small, and the precision is enough for millimetre measurements, statistical values or angles. The numeric values were encoded in little-endian (LE) format (Cohen, 1980) that appeared consistent across all environments but should be explicitly adhered to if other components were to be added to the application.

The message schemata were described in JSON files to make them available across languages. Shown in Listing 4.1 is a simple example message schema for a nanosecond timestamp (*t_ns*), *type* and one or more 3D *points* stored as floats.

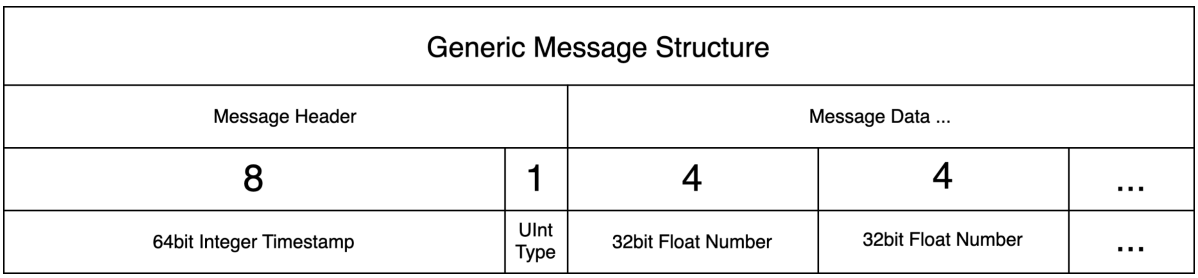

Figure 4.4.: The basic message structure for transmitting numeric sensor readings

The root object's property names resolve to the key under which the value could later be accessed. The *index* property specifies the byte index in the message, and *count* specifies if the value repeats in sequence or is singular. *dims* defines the dimensions for the value (e.g. '3' for a 3D point). The *type* could be a 'Uint8', 'Float32' or a 'BigInt64' and the property 'le' specifies if this value is encoded as LE.

```
{
  "t_ns": {
    "index": 0,
    "count": 1,
    "dims": 1,
    "type": "BigInt64",
    "le": true
  },
  "type": {
    "index": 8,
    "count": 1,
    "dims": 1,
    "type": "Uint8",
    "le": true
  },
  "points": {
    "index": 9,
    "dims": 3,
    "type": "Float32",
    "le": true
  }
}
```

Listing 4.1.: Example pose message schema

4.7. Application components

The application comprises several third-party components merely deployed as-is (WebRTC, databases, static web server) and the custom-developed parts described in the following section.

4.7.1. Core SDK module

The core functionality was built into a separate module to enable integration into other setups using different frameworks or architectures. The module uses the NPM's package format and can be utilised in the browser and Node.js. While this module carries the most fundamental functionality, it was created last in the development process, as the essential parts only crystallised during the initial development phase.

4.7.2. Web frontend

The web frontend provides the main entry point for the users. It allows authentication via a local username and password combination and provides objects modelled as virtual 'spaces' that are the central anchor to organise all communications, as defined in Figure 4.3. Users can create spaces, name them and then join them, becoming active data producers, or choose to view them as passive spectators.

Depending on the participant's role, a space is rendered as a different set of components. Participants who actively join have access to a LocalProducer and a HeadTracker component. These components provide a direct link via WebSockets to the external data-producer utilities and a WebBluetooth connection to the custom head-tracking device built on Arduino. Those who only view the space do so via a dedicated scene viewer component that brings together all incoming streams and signals.

The frontend was designed to coordinate the connections between the WebRTC server, the backend API and the local utilities. It also implements the various web standard APIs needed for sound, graphics and communication.

4.7.3. API backend

In the backend, the API server was tasked with managing the basic connecting objects (spaces and users), general authentication, and generation of access tokens for the LiveKit server. Through its real-time implementation, it can notify connected clients of changes like other connecting users or updates to data. The Feathers framework exports its own client library that is specifically generated for the current server configuration. It can be directly integrated into Vue using a client adapter module that handles authentication and basic create, retrieve, update and delete (CRUD) operations.

4.7.4. Native utilities

Three different native utilities were additionally implemented.

The general *data producer* was set up as a CLI utility in Python, implementing various Python-specific extensions: the DepthAI framework[2], used to work with the Luxonis Oak-D line of 3D-cameras, Intel's OpenVino[3] for interacting with various machine learning (ML) models for pose recognition or point-cloud extraction, Open3D (Zhou et al., 2018) for working with point cloud data and general spatial operations and PyMotion (Ponton, 2023), a library for working with recorded Biovision hierarchy (BVH) motion capture data files. Python also allowed for easy statistical data analysis using NumPy, which was used to perform movement quality extraction.

For real-time streaming of live motion capture data from the Captury Live system, there currently only exists a C++ client library provided by the system's manufacturer. Thus, the *Captury data producer* component had to be implemented separately and uses a C++-based WebSockets server streaming the library's received data.

[2] https://github.com/luxonis/depthai
[3] https://github.com/openvinotoolkit/openvino

As there was no affordable, open and platform-independent *head-tracking* solution, this component was quickly prototyped using a BluetoothLE-ready Arduino device (Nano RP2040 Connect) and an IMU component for absolute orientation measurement by Adafruit (9-DOF Absolute Orientation IMU Fusion Breakout) that can be directly connected to the Arduino using the Inter-Integrated Circuit (I2C) serial bus. The data read from the IMU device is then posted as binary messages on a simple Bluetooth service. The device could be directly integrated using the browser's WebBluetooth web standard.

5. Implementation

5.1. Project Setup

The underlying *server infrastructure* for the reference implementation was an existing server with 16 CPU-cores (with multithreading), 64 gigabyte (GB) random access memory (RAM) and a 512 GB solid-state drive (SSD) drive, located on the Johannes Gutenberg University Mainz (JGU) campus and connected to the internet via a dedicated one-gigabit network connection (see speedtest[1] results in Listing 5.1).

```
Server: Deutsche Glasfaser - Frankfurt (id: 54504)
ISP: Johannes Gutenberg-Universitaet Mainz
Idle Latency:     1.14 ms   (jitter: 0.05ms, low: 1.07ms, high: 1.27ms)
Download:   925.22 Mbps (data used: 417.1 MB)
2.78 ms   (jitter: 0.33ms, low: 1.32ms, high: 3.35ms)
Upload:   923.30 Mbps (data used: 416.7 MB)
6.59 ms   (jitter: 29.13ms, low: 1.18ms, high: 214.75ms)
```

Listing 5.1.: Speedtest: connection statistics for the server used to deploy the application

The orchestration was deployed first to allow development on a working remote WebRTC infrastructure. The basis was a clean, freshly bootstrapped Kubernetes installation running on the bare-metal server. LiveKit and its Redis database were installed via the application deployment manager Helm, using an official installation chart published by its

[1] https://www.speedtest.net/

maintainers[2]. To simplify the deployment, LiveKit was placed behind the reverse proxy Traefik[3] to manage secure sockets layer (SSL) termination via the LetsEncrypt[4] service and routing to the actual service running inside the cluster. However, this simplified setup required LiveKit to be configured to listen on a single TCP port instead of a range of UDP ports, as it would usually be deployed. The potential downside of this deployment configuration was deemed insignificant since the server only needs to service a handful of users. The detailed Kubernetes setup instructions are documented in the according folder in the project's repository[5].

The *LiveKit* installation was deployed with only a slight deviation from the default configuration. It was set up to use TCP as a transport protocol to allow easier integration with SSL termination using the reverse proxy. Otherwise, the configuration defined the endpoints for sending webhook requests and custom credentials for making requests to it via the server-side SDK and generating valid access tokens for users to connect to rooms. The Helm installation chart was used to set up the system along with its Redis database installation in a single command, and the server was immediately ready for connections.

The development process was conducted in a desktop environment, using the suite of tools developed by JetBrains (WebStorm, PyCharm and CLion), as these are free for educational use and provide a complete environment for development, including debugging, intelligent code completion, versioning, containerisation and deployment. A local Docker Desktop installation allowed running services and databases locally to support development before publishing to the production environment. Versioning was done via Git on the GitLab platform provided by the JGU[6].

[2] https://github.com/livekit/livekit-helm
[3] https://traefik.io/
[4] https://letsencrypt.org/
[5] Kubernetes setup instructions: https://github.com/dasantonym/sensorama/tree/master/kubernetes
[6] https://gitlab.rlp.net

5.2. API server

The first custom implementation, the API server, was generated using the Feathers CLI tool with standard username and password authentication and WebSockets, as well as HTTP transports, enabled. It provides the core services for Users, Spaces, Tokens and LiveKit events. These services were autogenerated using the Feathers CLI utility. They were used largely unmodified, except for adding the properties on the models for Users and Spaces as defined in section 4.6. A custom service class was added for the Tokens, as these do not persist in the database and instead are generated on the fly by the LiveKit server SDK. All other boilerplate code for the API, including the MongoDB integration, the authentication mechanism, and the REST and WebSockets transport integrations, were also autogenerated using the CLI. An additional custom Livekit event service was added to the API, allowing it to receive webhook requests via HTTP from the LiveKit server containing updates on connecting and disconnecting users. These events do not persist in the database but are relayed instead to the connected users via the real-time channels. The channels feature provided by Feathers was used to automatically subscribe, connecting users to updates on the services for Spaces and Livekit events.

5.3. Core SDK

The basic functionality was bundled in an NPM module to make the base code independent of the use case, which could later be used in other projects. This module contains the abstract classes ‘DataProducer’, ‘HeadTracker’ and ‘SonificationController’ alongside the message specification for the various types of transmitted data. It was written to propagate updates via events instead of the reactive patterns used in Vue so that it can also be used independently from the framework used for the study. Unit tests were added to the module to maintain a stable implementation.

5.4. User interface

The Quasar framework provides a CLI to generate new projects, which allows selecting basic implementation details (e.g. language, state management) and producing a complete and working empty Vue project with sample components that served as the starting point for the UI implementation. The first thing added to the project was the library 'feathers-pinia'[7], which is provided by the Feathers developer community and promises easy integration of an existing Feathers API with any Vue project using the Pinia[8] state management system used by Vue. The extension was integrated by linking the client library that the Feathers server project automatically generates by referencing the API's project folder and configuring basic authentication settings. The routing configuration and page components were set up according to the sitemap (Figure 5.1).

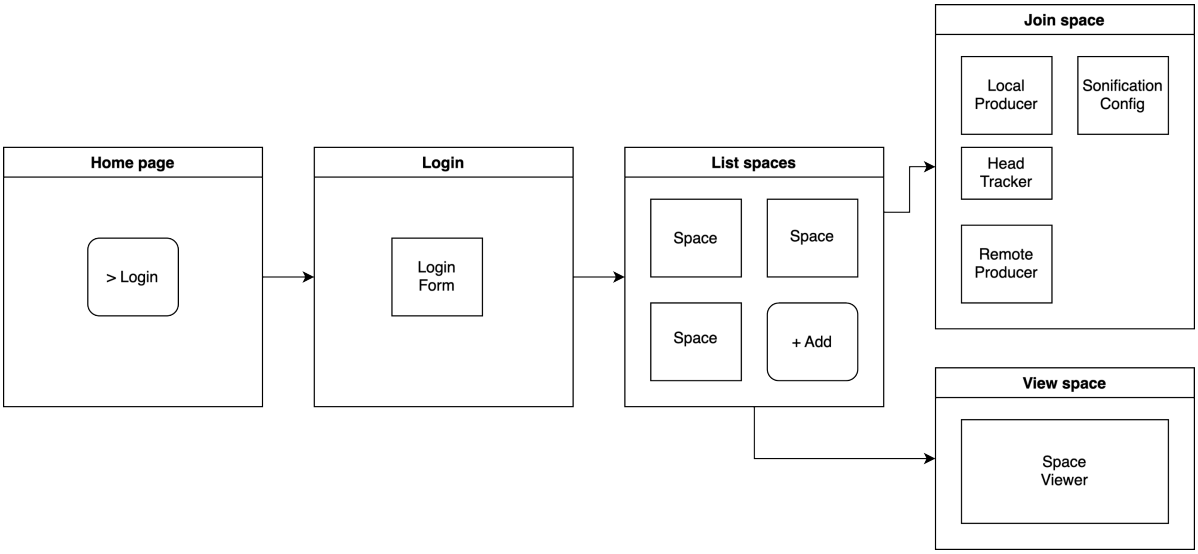

Figure 5.1.: Sitemap for the Sensorama UI

The overview page for the spaces presents a list of the names alongside the currently connected users and buttons to either join as a participant or passively view it as a spectator. On joining a space, the user first needs to activate their microphone to activate the audio context. The user is then presented with three basic control panels. The 'Data

[7] https://feathers-pinia.pages.dev
[8] https://pinia.vuejs.org/

producer' panel allows setting a URL of a local WebSockets server published by a data producer, setting the message type received from it, and optionally enabling a tracing function to log the packet transmission statistics. Once connected, the panel shows a preview of the incoming points data, transmission statistics and a button to disconnect. Internally, the panel creates a reactive data store that instantiates the 'DataProducer' class from the core SDK, watches incoming message events and populates the received data as reactive properties to be used across the UI in various other components without the need of creating additional class instances. The second panel, 'Head tracker', works similarly but uses the Web Bluetooth API to select a nearby device and connect to it to receive its data messages. The third component configures the sonification by setting thresholds on incoming movement quality values. It connects to an instance of the 'SonificationController' class from the core SDK and allows the configuration of event thresholds, sound selection and tonal configuration.

When using the page to view a space, there is only one component, the 'SpaceViewer', that shows a 3D room in which all incoming points are rendered as small spheres, giving the impression of a human figure. For each participant, there is a differently coloured light source that follows the centre of mass of the points associated with the participant. The viewer also renders all sonifications and audio streams at their respective spatial positions. This allows the user to view the 3D scene on screen while listening to binaural audio or to use the built-in VR functionality to experience the scene in a visually immersive way.

5.5. Data producers

The general *data producer* was written in Python and provides multiple data sources: an interface to a Blazepose implementation on the Oak-D 3D camera, as well as reading depth images as point clouds from the camera and an interface to load and playback motion capture data in the BVH file format. All three data sources were implemented as separate

Python classes because the classes related to the Oak-D camera were built by modifying existing code for pose recognition (geaxgx, 2021) and point cloud processing (Erol444, 2022). The BVH data source was created from scratch and implemented to facilitate testing and development by using playback of motion capture data of professional dancers pre-recorded on the Captury Live system. All data source classes were set up to support calling a function in a loop and returning current point data as a multidimensional array. The returned data is packed as a byte sequence and sent over WebSockets in the appropriate format (see section 4.6). This way, the disparate sources could be imported into a single central file that uses a combined set of utility functions for running a WebSockets server and packing data. Due to the lack of a Python-based client for the Captury Live system, the Captury producer had to be implemented separately as a C++ project using CMake[9] as a build system and based on the 'RemoteCaptury' client library (The Captury GmbH, 2018), as well as an example project for a WebSockets server implementation in C++ (Rehn, 2016).

The custom-built head-tracking device was implemented as an Arduino project. As such, it was first realised as a hardware setup and then outfitted with custom firmware written in the Arduino-specific flavour of C/C+. The hardware implementation was created using the 'Arduino Connect RP2040'[10], which is based on the Raspberry PI 2040 microcontroller and has an onboard BluetoohLE module. The IMU module used was the '9-DOF Absolute Orientation IMU Fusion Breakout'[11] which uses the BNO055[12] chip produced by Bosch. This chip has the benefit of already pre-processing the data from the gyroscope, accelerometer and magnetometer into an absolute world position that can be directly read from the breakout board via the I2C bus. As a third component, a small 3.7V lithium battery was added alongside a charging module[13]. Only six connections needed to be soldered between the three modules (2x charger and 4x IMU), and the

[9] https://cmake.org/
[10] https://docs.arduino.cc/hardware/nano-rp2040-connect/
[11] https://www.adafruit.com/product/4646
[12] https://www.bosch-sensortec.com/products/smart-sensor-systems/bno055/
[13] https://www.adafruit.com/product/1905

resulting circuit was ready to function as a custom head tracker (Figure 5.2). For the software implementation, the basic example code for the Adafruit module was used to set up continuous polling of the positioning module, reading the values for position and device calibration status and sending them as byte sequences at a fixed rate of 25fps over BluetoothLE.

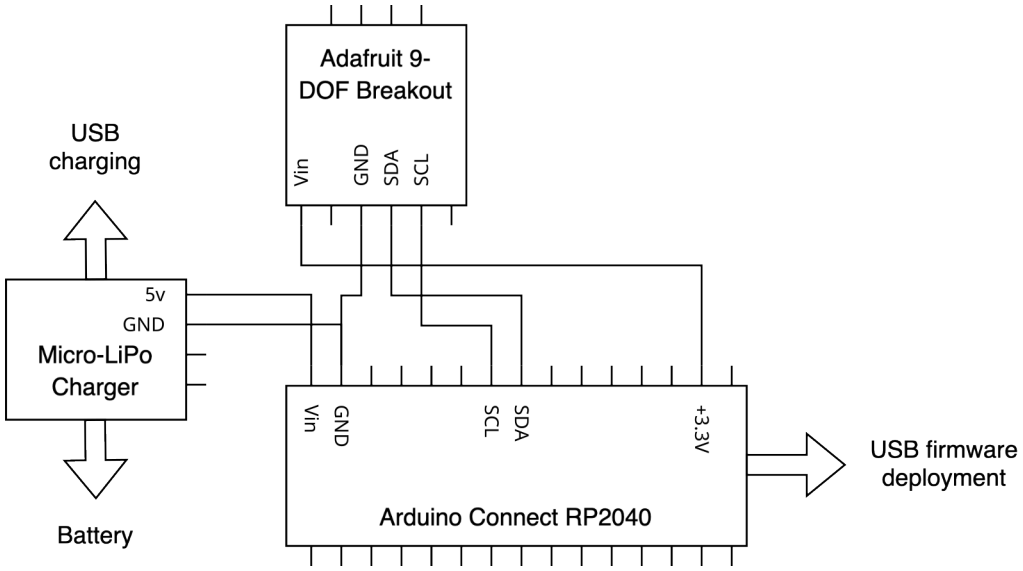

Figure 5.2.: Wiring diagram for the custom head tracker

6. Discussion

6.1. Application performance

The statistical evaluation focused both on the extrinsic properties of the application and the development process itself (e.g. code volume, complexity, time spent on development), as well as the intrinsic functional properties of the core functionality that are reflected in the cost of CPU-, network-usage and message latency. The cost of rendering audio and video, as well as general user experience metrics, were beyond the scope of the study, as these are highly specific to the task being implemented and are considered transient.

Two computers used in the performance evaluation were end-user laptops with the following specifications:

Computer A was equipped with a dual-core Intel Core i5 CPU, 16 GB of RAM and a 500 GB SSD. *Computer B* was equipped with an Apple M1 Max CPU with ten cores, 64 GB of RAM and a 1 terabyte (TB) SSD.

The test systems were located in a rural area 80 kilometres (kms) outside Mainz, used the Google Chrome browser (version 121.0.6167.85), ran macOS 13.6.4 and connected to a consumer asynchronous digital subscriber line (ADSL) connection (60 megabit (Mbit) download and 15 Mbit upload) via an 802.11ac wireless connection (Listing 6.1).

Latency was measured by time stamping each message in the data producer and then logging the arrival time in the local and the remote browser. Both computers synchronised

```
      Server: Deutsche Glasfaser - Kaiserslautern (id: 53558)
         ISP: Inexio
Idle Latency:    25.60 ms   (jitter: 0.08ms, low: 25.41ms, high: 25.73ms)
    Download:    58.02 Mbps (data used: 57.7 MB)
                 43.56 ms   (jitter: 1.87ms, low: 24.36ms, high: 47.92ms)
      Upload:    14.19 Mbps (data used: 11.4 MB)
                 26.30 ms   (jitter: 0.93ms, low: 23.86ms, high: 33.05ms)
```

Listing 6.1.: Speedtest: connection statistics for the test clients

their clocks using the same Network Time Protocol (NTP) server on the local netork. The synchronisation was refreshed every 10 seconds during the data sampling and provided sub-millisecond clock accuracy throughout the measuring process. The measurements ran for ten minutes each and always used the BVH data producer, sending the message type for movement qualities alongside 29 key points at a rate of 25 messages per second. The payload size was 453 bytes for each message, amounting to a required bandwidth of about 11.3 kilobyte per second (KB/s) for each motion capture data stream. They were repeated ten times to account for overall network service quality variance. The audio streams were published alongside the data packets but were not measured for latency. The CPU, RAM and bandwidth usage for LiveKit on the server was recorded as well. The results of the latency analysis are shown in separate graphs for each computer containing an average across all datasets for the local and remote messages on each device.

Results for computer A (Figure 6.1) show an overall average latency of approx. 30 millisecond(ms) for remote messages received over the WebRTC connection and approx. 2 ms for the connection from the local data producer to the browser. The averages show a standard deviation of approx. 1.4 ms for the remote connection and about approx. 0.2 ms for the local connection.

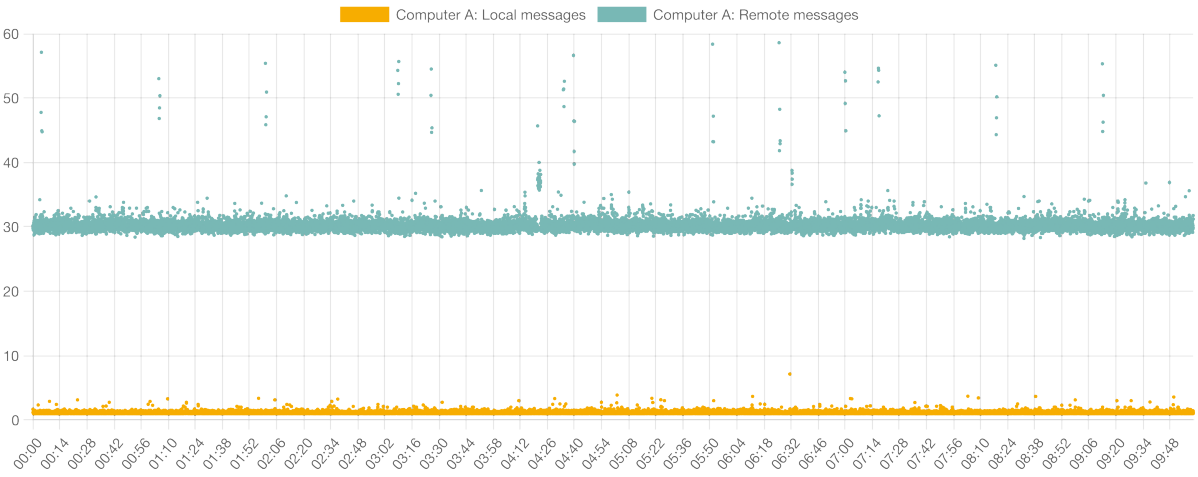

Figure 6.1.: Computer A: Average latency in milliseconds for local and remote messages

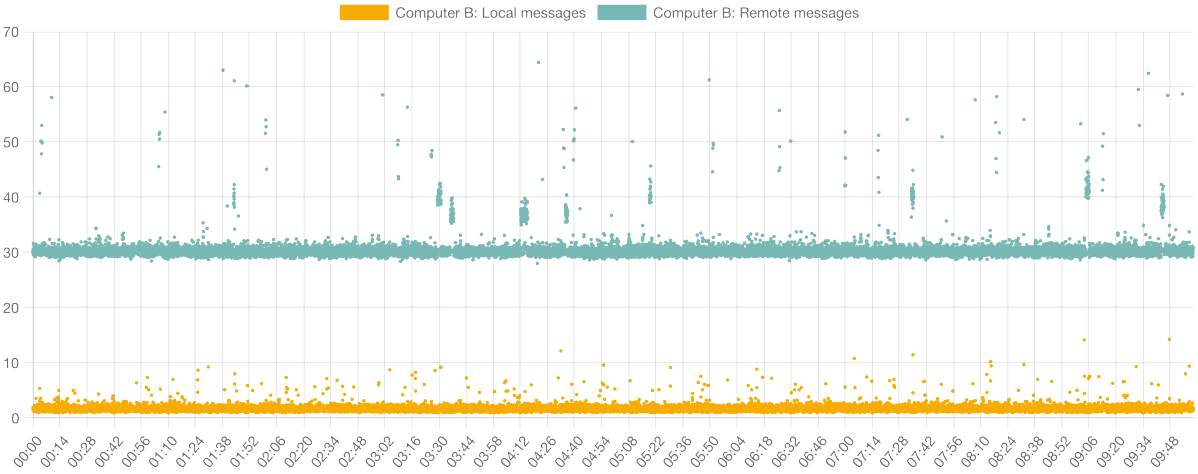

Figure 6.2.: Computer B: Average latency in milliseconds for local and remote messages

The results gathered on computer B (Figure 6.2) also show an average latency of approx. 30 ms for the remote messages and approx. 1.7 ms for the local data producer connection. Here, the standard deviation is approx. 2.1 ms for the remote connection and approx. 0.6 ms for the local connection.

On the server, the average overall network consumption for the LiveKit server was at approx. 106 KB/s for incoming and approx. 156 KB/s for outgoing traffic, moderately fluctuating with a peak of approx. 122 KB/s incoming and approx. 198 KB/s, while the isolated packet traffic remained relatively stable at an average of approx. 32 KB/s (in) and approx. 33 KB/s (out) with respective maxima of approx. 35 KB/s and ap-

prox. 36 KB/s (Figure 6.3). The server compute resource usage was minimal with only an average of approx. 0.04 CPU cores and approx. 81 megabyte (MB) RAM usage (Figure 6.4).

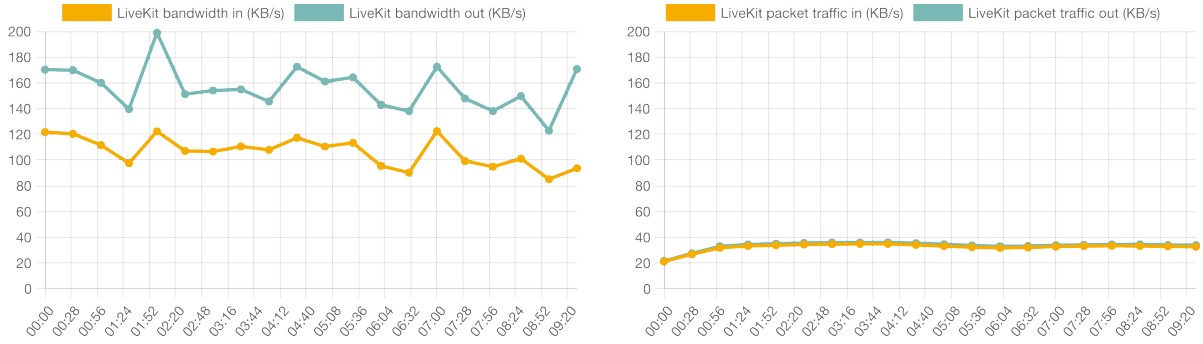

Figure 6.3.: Average network usage for the LiveKit server during tests (KB/s)

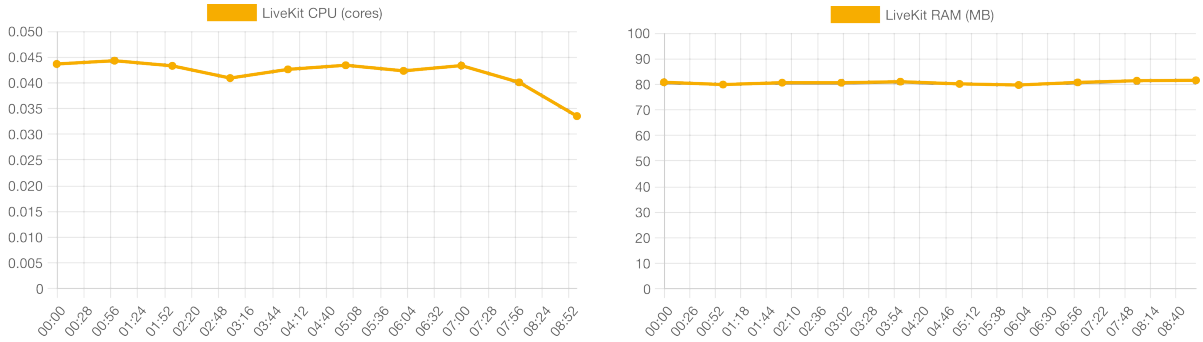

Figure 6.4.: Average CPU and RAM usage for the LiveKit server during tests

Overall, the performance analysis provided a very positive result as even the message latency for the home network connection on a moderately equipped laptop (Computer A) remained at around 30 ms, which is well below the lower tolerance limit of 160 ms proposed in a study by Hopkins et al., 2022 on tolerable audio latency for live music creation. On the server side, the impact on the computing resources was barely registering, the network throughput left significant headroom for more connections over the server's a gigabit connection and would most likely work as well on a 100 Mbit connection and a server with fewer cores than the test setup.

Another pairing of test computers was attempted using a local Ethernet connection on campus at Mainz University of Applied Sciences, but was discarded, because latencies

were in the single digit to sub-milliseconds range and thus could not be properly measured due lack of a precision clock system.

6.2. Workload analyis

The evaluation of time spent on the development work was based on the timesheets kept during the process. Only the time spent on the actual programming and infrastructure creation was tracked, as the research could not be adequately separated from the work on the study itself. All recorded tasks were categorised by the language used (e.g. JavaScript versus Python), the component worked on (e.g. UI vs. API) and the type of work (e.g. programming versus administration).

In total, approx. 144 work hours were spent creating the reference implementation and its deployment on the test infrastructure. This would amount to approx. 18 workdays or three and a half five-day workweeks, assuming eight hours for each day, which would be in line with §3 of the German law for labour time regulation, known as the 'Arbeitszeitgesetz' (see (Bundesamt für Justiz, n.d.)).

The distribution of time spent on software development versus documentation and setting up the infrastructure (administration) and constructing the head-tracker's hardware implementation (Figure 6.5) clearly shows that the largest amount of time (approx. 93%) was spent on programming and the latter two factors were only marginal in the effort needed to be put in.

Of the amount of time spent in software development, the time spent by programming languages (Figure 6.6) shows a clear majority of approx. 72% for JS, given that it was chosen as the language for the UI and the API and the project's general primary language. However, Python still takes up approx. 21% of the work hours spent on programming. C++ required only a marginal amount of work with approx. 7% of the time spent.

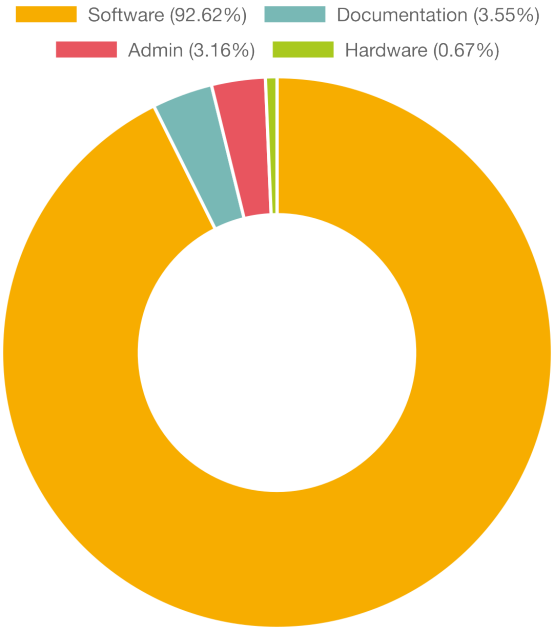

Figure 6.5.: Distribution of time spent on development by work area, language and application component

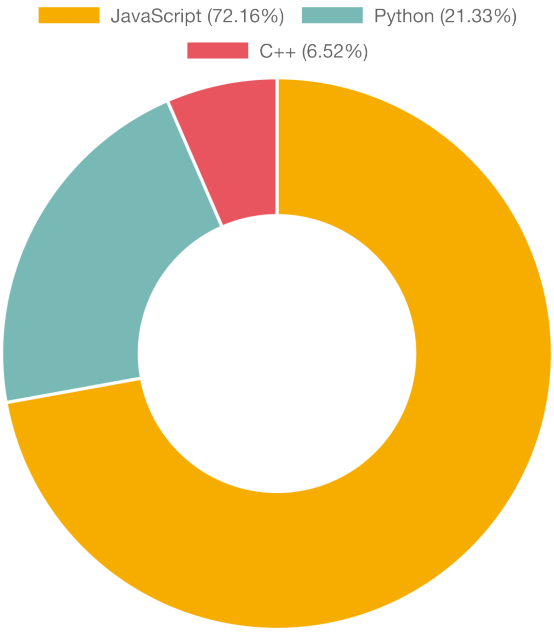

Figure 6.6.: Distribution of time spent on development by programming language

Time spent on the different components (Figure 6.7) shows that about half of the time was spent on the user interface (approx. 56%). The component using the second largest

amount of time is the Python data producer (approx. 20%). All other components remained under 10% and notably, the SDK required only 7.3% of time as it was mostly constructed by refactoring existing code and adding tests. Also noteworthy is the time spent on the API server, which was only approx. 4% since it was mainly autogenerated and required only very litte modification. Equally minimal is the time spent on the deployment, also with only approx. 4%.

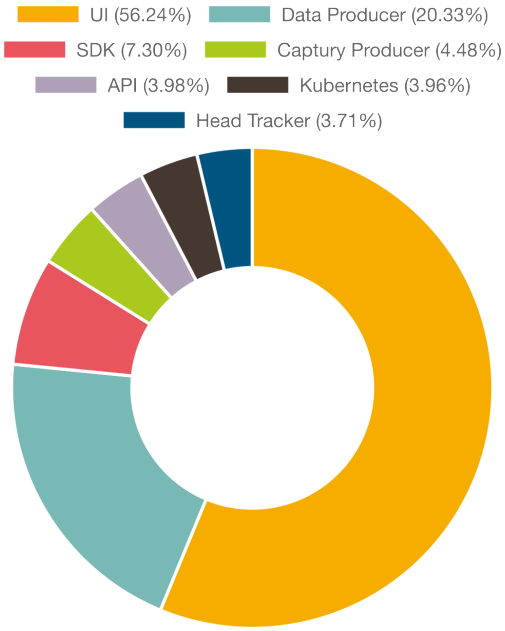

Figure 6.7.: Distribution of time spent on development by programming language

The overall time spent of about a month should be seen in the context of an ongoing larger project and on the basis of parts of the application being meant for reuse. If the API and data producers would be kept largely stable and only the UI would be subject for modifications, extensions or even complete replacement, this would affect roughly half of time spent on the initial development. While an accurate prediction on the time spent on subsequent modifications cannot be made, it should equally stay under a month and thus be an approprate time investment to support future projects.

6.3. Code quality

Code quality was analysed using the service SonarCloud[1], which is free for small projects and provides an extensive set of metrics, of which the code weight (files, lines of code) and complexity were of specific interest, as defined in chapter 3. The default recommended limit for cognitive complexity set by the SonarCloud code analysis setup is 15 for JS and Python and 25 for C++. The maximum cognitive complexity (Figure 6.8) shows a significant overhead of 45 for the core SDK, 77 for the data producer and 41 for the Captury producer. These extremely high maxima were caused by the ruke-based sonification controller class in the SDK, which relies heavily on if-clauses and could be subject to further refactoring. In the data producer, the extremely high maximum of complexity is caused by a file pulled in from example code being used to build the Oak-D camera integration and the captury producer was quickly put together as a proof of concept. Still, the median values all remain under the limit and show a rather moderate cognitive complexity concentrated in the producers and the head tracker. Notably, the API has a median complexity of zero, which is due to the aspect-oriented design that does not rely on control structures. The UI shows a very low median complexity of one, with a tolerable maximum of 21, which was the desired outcome to keep this part more easily hackable and easy to grasp.

In terms of weight measured by the number of files, the most code files were produced for the UI and the API, which aligns with the distribution of functionality according to the concept. The distribution of weight regarding the total lines of code without comments produced for each component aligns with the number of files for the UI but shows a significant overhead for the data producer with almost as many lines of code distributed among a third of the number of files (Figure 6.9). It generally shows a relatively moderate weight for the entire application, with a maximum of 34 files created for the UI and a total of 83 files across all components. However, the data producer presents an outlier in this regard, because with 1261 it uses almost as many lines of code as the UI, while

[1] https://sonarcloud.io

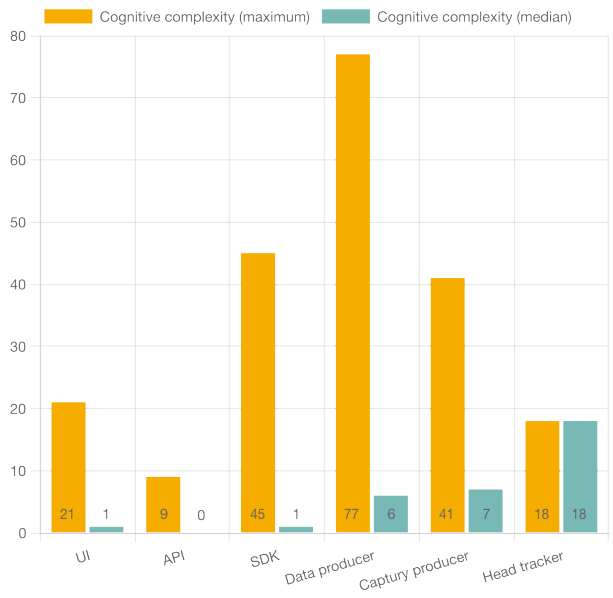

Figure 6.8.: Cognitive complexity per component

consisting of a little over a third of total code files (13 versus 34). This can be explained by the external code used as a basis for the data producer, which used several files built as more monolithic structures which weren't part of the refactoring cycle. Still, when looking at the maximum versus median amount of lines of code per file (Figure 6.10), it becomes more apparent, that the concentration of code weight is due to only relatively few files, as all median values remain within the alotted maximum of under 160-200 lines (see chapter 3). Here, it becomes even more apparent that the data producer using the external code presents an outlier to the general coding norms used throughout the project.

Summarily, the results for the code quality of such a somewhat non-standard development process provide further encouragement to pursue more projects based on the same strategy or even on the study's resulting source code. The average complexity and weight of the code, along with the documentation and comments added, should enable third-party developers to engage with the results by reviewing and modifying specific parts to introduce their own experimental data capture, visualisation or sonification methods.

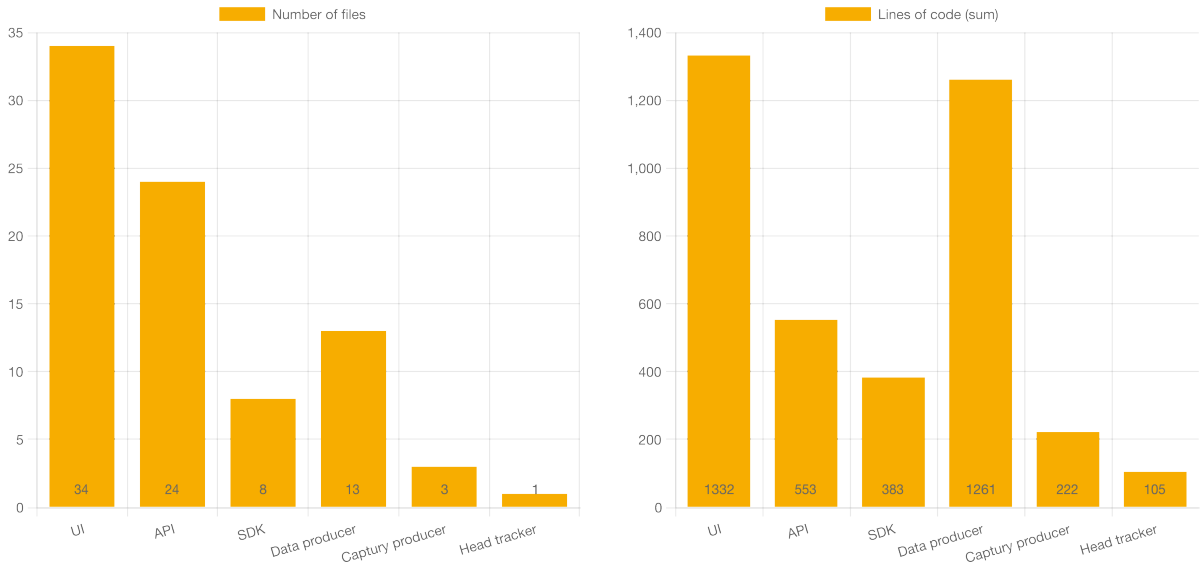

Figure 6.9.: Number of source files and total lines of code per application component

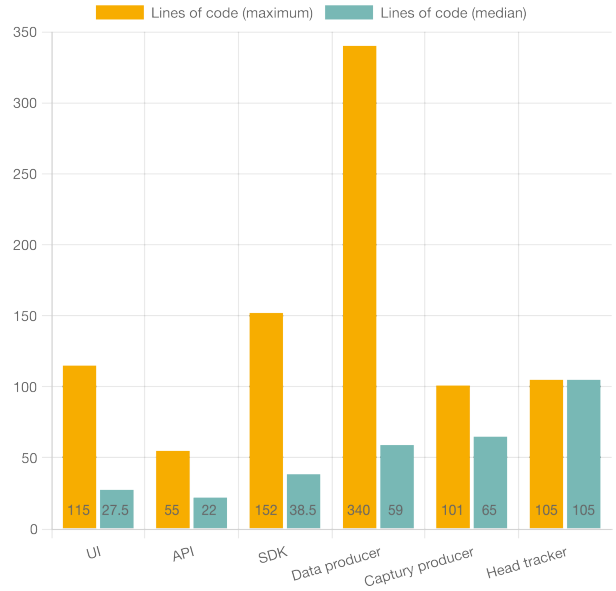

Figure 6.10.: Maximum and median number of lines of code per application component

6.4. Critical evaluation

The development process followed the guiding principles defined in the application concept (see chapter 4). It was a pleasant experience with the selected frameworks delivering on their promised functionality and ease of use. The initial setup was quick and simple

due to the ease of setup of the LiveKit WebRTC server and the generation of boilerplate code for the API and the UI. A development environment was set up equally quickly, and work immediately started with practical experimentation, providing a motivational boost by quickly establishing tangible results. Web standards integration was simple and efficient through directly implemented standards such as WebSockets and WebBluetooth and the other ones integrated through third-party libraries. However, the implementation of the web audio standard still leaves a lot to desire, especially the support for customised spatial audio in the browser. Currently, there is no built-in way to load custom HRTF data, which would drastically improve the accuracy of spatial positioning for sound. There are approaches using a custom build of the Chromium browser (Pike et al., 2015) or a custom audio node (Carpentier, 2015), which unfortunately does not work with the spatially oriented format for acoustics (SOFA) file format, and due to the study's time constraints did not make it into the reference implementation.

The selection of appropriate tools and libraries might be a challenge for many beginners with basic programming knowledge who are not primarily working in web development. As the libraries and frameworks that make up the broad spectrum of available web development technologies tend to favour different paradigms that, in turn, are also subject to frequent change due to trends and 'hype cycles', it can be challenging to keep up and daunting to make an informed choice among the available options. Looking at the options evaluated throughout this study, there are a few factors to consider when deciding which framework to prefer for a specific type of project. It is vital to decide if a project will be maintained in the long run. If, as in this case, the implementation is transient and does not require long-term maintenance, then the choice should be guided by the provided feature set and the paradigms implemented. While it is always a good choice to use a library with a large community and a longstanding presence, a more experimental and niche case might require something newer or less popular.

For the development process itself, the strategy of getting something up and running as quickly as possible, working on that while regularly refactoring and restructuring, was a gratifying experience. While this approach might be perceived as not exactly

team-friendly due to the danger of conflicting work, it can still be pursued in very small teams by agreeing on basic protocols and then working on separate components. The partitioning of the application into several components was also highly beneficial because it allowed for different languages to be used where appropriate. There are many more implementations of movement analysis for Python than for JavaScript or Node.js. Before porting functionality to a different language, it is easier to set up a separate microservice or tool and use a standardised messaging protocol to communicate with the rest of the application. Another benefit of the partitioning design pattern is that components can be discarded and replaced with different technologies and supporting libraries without necessarily influencing the application's overall functionality. It is important to note, though, that while the UI is thought of as transient and unstable, unit testing is still highly recommended for the stable components that should keep being used outside of the specific implementation. While it could be beneficial to develop the core SDK using test-driven development and, once stable, working on the implementation, it would add a period during which there would be only theoretical planning and development. This seems out of place for small creative projects that work interdisciplinary, like a dance production that needs its own remote collaboration tool and that needs to get started as quickly as possible. Nonetheless, it is still important to have a finalising phase in which the core functionality is extracted, documented and outfitted with tests to keep the SDK as an artefact on which to base further projects.

7. Conclusion and outlook

The survey of available technologies and methods in chapter 2 showed a broad range of readily available options. Based on the methodology established in chapter 3, a reference implementation could be created (chapter 4 and chapter 5) and its discussion in chapter 6 produced a positive recommendation for such a ‘single-use’ or ‘ephemeral’ development. This seems especially valuable in cases without a focus on commercial deployment and exploitation of services or products, but rather specialised tools that are an intrinsic part of smaller self-contained and short-lived projects. Additionally, the resulting reusable components could significantly accelerate future development if applied roughly in the same area of usage.

Regarding the bulk of the application as transient and extracting the core functionality into well-documented and tested modules further has the benefit that if the application is passed on to other developers for another project or task, they can decide to either go with the existing base infrastructure or to take only the core functionality and to implement it in their own favoured environment. It enables them to port the core to other languages more easily. In the web development area, this is essential because, while the standards and the application's feature set might stay the same, the framework and tooling landscape certainly doesn’t, and even just a few years can render the application obsolete if it is not constantly maintained.

Funding schemes for most digital projects in niche culture or arts disciplines usually do not allocate sums that would allow for more than one or two developers to be included on the team. Often, the people working on these projects are not formally trained software

engineers, but rather creative coders, hackers or engaging from a multidisciplinary angle and for whom the technological aspects came as a secondary interest to a primary education in arts or other creative fields. This suggests approaching software development from a different angle and valuing the development process over the actual result. While the objective is always to produce a functional implementation, it is less critical to insist on clean code or engineering virtuosity such as scalability, maintainability, and perfect abstraction. It aims to *decommodify the notion of an application* and to turn it into an ephemeral *statement within a dialogue between engineering and artistic practice.*

We propose the expression of *code composting* to describe a cyclical development process that alternates between intuitive composition focused on loose experimentation and analytical decomposition using reflection and refactoring to extract essential structures emergent in the compositional process. The extracted components then become a *substrate* for subsequent development processes, repeating the cycle and producing additional building blocks. Here, the focus shifts away from traditional coding virtuosity and can help to lower the barrier between engineers and artists. This should enable a more intertwined and participatory dialogue between all project's participants that is not just expressed in development meetings but actual proposed hacks and modifications or additions, given the interest on behalf of all parties involved.

Code composting values *processes over products* and embraces the *ephemerality of contemporary digital technologies* as intrinsic design factors. Thus, development should focus on constant reflection and re-evaluation of development results and be ready to potentially discard everything but a seed for a fresh start while keeping the knowledge and insights gained in the process alive through continuous inclusive, transdisciplinary dialogue and experimentation. It positively values what otherwise would be negatively termed as ‘rot’, the result of a messy growth process in an amalgamation of engineering and artistic experimentation.

In general, this development method can explicitly be recommended for experimental, non-commercial endeavours where factors like scale or long-term maintainability are secondary and where development should happen from within the actual practice instead of trying to communicate wishes and needs to external service providers.

Bibliography

A-Frame. (2015, September 15). *Web framework for building virtual reality experiences.* Retrieved January 25, 2024, from https://github.com/aframevr/aframe

Babylon.js. (2013, June 27). *Babylon.js is a powerful, beautiful, simple, and open game and rendering engine packed into a friendly javascript framework.* Retrieved January 25, 2024, from https://github.com/BabylonJS/Babylon.js

Barbier, J. (2014, June 9). *It's here: Docker 1.0.* Retrieved January 15, 2024, from https://web.archive.org/web/20220518024454/https://www.docker.com/blog/its-here-docker-1-0/

Barr, M. (2015, October 8). *Embedded systems glossary.* Retrieved January 20, 2024, from https://barrgroup.com/embedded-systems/glossary

Barragán, H. (2022, June 16). *The untold history of arduino.* Retrieved January 20, 2024, from https://arduinohistory.github.io/

Bazarevsky, V., Grishchenko, I., Raveendran, K., Zhu, T., Zhang, F., & Grundmann, M. (2020). Blazepose: On-device real-time body pose tracking.

Brandl, R. (2023, January 4). *The Most Popular Video Call Conferencing Platforms Worldwide.* Retrieved April 3, 2023, from https://www.emailtooltester.com/en/blog/video-conferencing-market-share/

Brock, H., Schmitz, G., Baumann, J., & Effenberg, A. O. (2012). If motion sounds: Movement sonification based on inertial sensor data [ENGINEERING OF SPORT CONFERENCE 2012]. *Procedia Engineering*, *34*, 556–561. https://doi.org/https://doi.org/10.1016/j.proeng.2012.04.095

Bundesamt für Justiz. (n.d.). *Arbeitszeitgesetz (arbzg) § 3 arbeitszeit der arbeitnehmer*. Retrieved February 4, 2024, from https://www.gesetze-im-internet.de/arbzg/__3.html

Burns, B. (2018, July 20). *The history of Kubernetes & the community behind it*. Retrieved January 15, 2024, from https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/

Buschmann, F., Henney, K., & Schmidt, D. (2007). *Pattern-oriented software architecture: A pattern language for distributed computing*. John Wiley & Sons, Inc.

Campbell, G. A. (2023, August). *Cognitive complexity: A new way of measuring understandability* (tech. rep.). Retrieved February 4, 2024, from https://www.sonarsource.com/resources/cognitive-complexity/

Carpentier, T. (2015, January). Binaural synthesis with the web audio api. In S. Goldszmidt, N. Schnell, V. Saiz & B. Matuszewski (Eds.), *Proceedings of the international web audio conference*. IRCAM.

Cohen, D. (1980, April). *On holy wars and a plea for peace* (IEN No. 137). RFC Editor. RFC Editor. https://www.rfc-editor.org/ien/ien137.txt

Couriol, B. (2021, April 7). *10 years after inception, WebRTC becomes an official web standard*. Retrieved April 3, 2023, from https://www.infoq.com/news/2021/04/webrtc-official-web-standard/

Daigle, K., & GitHub. (2023, November 8). *Octoverse: The state of open source and rise of AI in 2023*. Retrieved January 14, 2024, from https://github.blog/2023-11-08-the-state-of-open-source-and-ai/

Davis, J., Nguyen, T., & Simmons, J. (2023, February 1). *Interop 2023: Pushing interoperability forward*. Retrieved April 3, 2023, from https://webkit.org/blog/13706/interop-2023/

Deveria, A. (2024, January 23). *Can i use: Websockets*. Retrieved January 25, 2024, from https://caniuse.com/websockets

doppp & forum users. (2019, October 2). *Hackernews: Tiny acquires Meteor*. Retrieved January 17, 2024, from https://news.ycombinator.com/item?id=21137653

Elementary. (2023, January 9). *Elementary is a javascript library for digital audio signal processing.* Retrieved January 25, 2024, from https://github.com/elemaudio/elementary

Erol444. (2022, March 9). *Depthai pointcloud generation.* Retrieved January 25, 2024, from https://github.com/luxonis/depthai-experiments/tree/master/gen2-pointcloud

Feathers. (2011, October 20). *The API and real-time application framework.* Retrieved January 25, 2024, from https://github.com/feathersjs/feathers

Fette, I., & Melnikov, A. (2011, November 1). *The WebSocket protocol.* Retrieved January 14, 2024, from https://datatracker.ietf.org/doc/html/rfc6455

Flocking. (2011, March 27). *Flocking - creative audio synthesis for the web.* Retrieved January 25, 2024, from https://github.com/lichen-community-systems/Flocking

Galli, M., Soares, R., & Oeschger, I. (2003, May 16). *Inner-browsing: Extending web browsing the navigation paradigm.* Retrieved January 11, 2024, from https://web.archive.org/web/20030810102320/http://devedge.netscape.com/viewsource/2003/inner-browsing/

Garrett, J. J. (2005, February 18). *Ajax: A new approach to web applications.* Retrieved January 11, 2024, from https://web.archive.org/web/20150910072359/http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

geaxgx. (2021, March 12). *Blazepose tracking with DepthAI.* Retrieved January 25, 2024, from https://github.com/geaxgx/depthai_blazepose

Gigante, M. A. (1993). 1 - virtual reality: Definitions, history and applications. In R. Earnshaw, M. Gigante & H. Jones (Eds.), *Virtual reality systems* (pp. 3–14). Academic Press. https://doi.org/https://doi.org/10.1016/B978-0-12-227748-1.50009-3

Google. (n.d.). *Are the webrtc components from google's acquisition of global ip solutions?* Retrieved January 29, 2024, from https://web.archive.org/web/20110607005550/https://webrtc.org/faq/#TOC-Are-the-WebRTC-components-from-Goog

Greif, S., & Burel, E. (2022, February 16). *The state of JS: Most used backend frameworks in 2021.* Retrieved January 17, 2024, from https://2021.stateofjs.com/en-US/libraries/back-end-frameworks/

Greif, S., & Burel, E. (2023a, January 11). *The state of JS*. Retrieved January 11, 2024, from https://stateofjs.com

Greif, S., & Burel, E. (2023b, January 11). *The state of JS: Most used backend frameworks in 2022*. Retrieved January 11, 2024, from https://2022.stateofjs.com/en-US/other-tools/#backend_frameworks

Greif, S., & Burel, E. (2023c, January 11). *The state of JS: Most used frontend frameworks in 2022*. Retrieved January 11, 2024, from https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/

Heilig, M. L. (1962, August). Sensorama simulator.

Hermann, T., Hunt, A., & Neuhoff, J. G. (Eds.). (2011). *The Sonification Handbook*. Logos Publishing House. https://nbn-resolving.org/urn:nbn:de:0070-pub-24423492,%20https://pub.uni-bielefeld.de/record/2442349

Holowaychuck, T. (2009, June 26). *Fast, unopinionated, minimalist web framework for node.* Retrieved January 25, 2024, from https://github.com/expressjs/express

Hopkins, T., Weng, S. C.-C., Vanukuru, R., Wenzel, E., Banic, A., & Do, E. Y.-L. (2022). How late is too late? effects of network latency on audio-visual perception during ar remote musical collaboration. *2022 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, 686–687. https://doi.org/10.1109/VRW55335.2022.00194

Howler.js. (2013, January 28). *Javascript audio library for the modern web.* Retrieved January 25, 2024, from https://github.com/goldfire/howler.js

IBM. (n.d.-a). *What is container orchestration?* Retrieved January 15, 2024, from https://www.ibm.com/topics/container-orchestration

IBM. (n.d.-b). *What is containerization?* Retrieved January 15, 2024, from https://www.ibm.com/topics/containerization

Iyengar, M. (2021, March 10). *WebRTC architecture basics: P2P, SFU, MCU, and hybrid approaches*. Retrieved January 29, 2024, from https://medium.com/securemeeting/webrtc-architecture-basics-p2p-sfu-mcu-and-hybrid-approaches-6e7d77a46a66

Jackson, J. (2012, October 1). *Microsoft augments Javascript for large-scale development*. Retrieved January 15, 2024, from https://web.archive.org/web/20131217223751/

http://www.cio.com/article/717679/Microsoft_Augments_Javascript_for_Large_scale_Development

Janus Gateway. (2014, February 11). *Janus webrtc server*. Retrieved January 25, 2024, from https://github.com/meetecho/janus-gateway

Jasmine. (2008, December 3). *Simple javascript testing framework for browsers and node.js*. Retrieved January 25, 2024, from https://github.com/jasmine/jasmine

Jest. (2013, December 10). *Delightful javascript testing*. Retrieved January 25, 2024, from https://github.com/jestjs/jest

Kendall, A., Grimes, M., & Cipolla, R. (2016). Posenet: A convolutional network for real-time 6-dof camera relocalization.

Krill, P. (2016, February 10). *Node.js foundation to shepherd Express web framework*. Retrieved January 14, 2024, from https://www.infoworld.com/article/3031686/nodejs-foundation-to-shepherd-express-web-framework.html

Kryski, E. (2016, April 6). *Why we built the best web framework you've probably never heard of (until now)*. Retrieved January 17, 2024, from https://blog.feathersjs.com/why-we-built-the-best-web-framework-you-ve-probably-never-heard-of-until-now-176afc5c6aac

Lardinois, F. (2019, October 2). *Tiny acquires Meteor*. Retrieved January 17, 2024, from https://techcrunch.com/2019/10/02/tiny-acquires-meteor/

Linux Foundation. (2015, July 21). *New Cloud Native Computing Foundation to drive alignment among container technologies*. Retrieved January 15, 2024, from https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-technologies/

Linux Foundation. (n.d.). *About the Open Container Initiative*. Retrieved January 15, 2024, from https://opencontainers.org/about/overview/

LiveKit. (2020, September 30). *End-to-end stack for webrtc. sfu media server and sdks.* Retrieved January 25, 2024, from https://github.com/livekit/livekit

LiveKit. (n.d.). *Livekit: About*. Retrieved January 19, 2024, from https://livekit.io/about

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Mediasoup. (2014, December 12). *Cutting edge webrtc video conferencings*. Retrieved January 25, 2024, from https://github.com/versatica/mediasoup

Mehta, S. (2023, August 17). *Microsoft's azure kinect developer kit technology transfers to partner ecosystem*. Retrieved January 15, 2024, from https://techcommunity.microsoft.com/t5/mixed-reality-blog/microsoft-s-azure-kinect-developer-kit-technology-transfers-to/ba-p/3899122

Meteor. (2012, February 1). *Meteor, the JavaScript app platform*. Retrieved January 25, 2024, from https://github.com/meteor/meteor

Minsky, M. (1980, June 1). *Telepresence*. Retrieved January 15, 2024, from https://web.media.mit.edu/~minsky/papers/Telepresence.html

Mocha. (2011, March 7). *Simple, flexible, fun javascript test framework for node.js & the browser*. Retrieved January 25, 2024, from https://github.com/mochajs/mocha

MongoDB. (2009, January 15). *The mongodb database*. Retrieved January 25, 2024, from https://github.com/mongodb/mongo

MySQL Server. (2014, September 26). *MySQL server, the world's most popular open source database, and MySQL cluster, a real-time, open source transactional database*. Retrieved January 25, 2024, from https://github.com/mysql/mysql-server

Nest. (2017, February 4). *A progressive Node.js framework for building efficient, scalable, and enterprise-grade server-side applications with TypeScript/JavaScript*. Retrieved January 25, 2024, from https://github.com/nestjs/nest

Netscape. (1995, December 4). *Netscape and Sun announce JavaScript*. Retrieved January 15, 2024, from https://web.archive.org/web/20070916144913/https://wp.netscape.com/newsref/pr/newsrelease67.html

Pike, C., Taylour, P., & Melchior, F. (2015, January). Delivering object-based 3d audio using the web audio api and the audio definition model. In S. Goldszmidt, N. Schnell, V. Saiz & B. Matuszewski (Eds.), *Proceedings of the international web audio conference*. IRCAM.

Ponton, J. L. (2023, June 2). *Pymotion: A python library for motion data*. Retrieved January 25, 2024, from https://github.com/UPC-ViRVIG/pymotion

PostgreSQL. (2010, September 21). *Mirror of the official PostgreSQL git repository*. Retrieved January 25, 2024, from https://github.com/postgres/postgres

Python Software Foundation. (n.d.). *History of the software*. Retrieved January 15, 2024, from https://docs.python.org/3/license.html

rafern. (2022, December 8). *Mediastream support #1634*. Retrieved January 29, 2024, from https://github.com/goldfire/howler.js/pull/1634

Rehn, A. (2016, December 3). *C++ WebSocket server demo*. Retrieved January 25, 2024, from https://github.com/adamrehn/websocket-server-demo

Rosenberg, L. H., Hammer, T., & Shaw, J. G. (1998). Software metrics and reliability. https://api.semanticscholar.org/CorpusID:59798472

Russell, A., & Berriman, F. (2015, October 8). *Progressive web apps: Escaping tabs without losing our soul*. Retrieved January 11, 2024, from https://medium.com/@slightlylate/progressive-apps-escaping-tabs-without-losing-our-soul-3b93a8561955

Skarbez, R., Brooks Jr., F. P., & Whitton, M. C. (2017). A survey of presence and related concepts. *ACM Computing Surveys*, *50*(96), 1–39. https://doi.org/10.1145/3134301

Socket.IO. (2010, March 11). *Realtime application framework (node.js server)*. Retrieved January 25, 2024, from https://github.com/socketio/socket.io

Stack Overflow. (2023a, June 13). *2023 Stack Overflow developer survey: Databases*. Retrieved January 25, 2024, from https://survey.stackoverflow.co/2023/#section-most-popular-technologies-databases

Stack Overflow. (2023b, June 13). *2023 Stack Overflow developer survey: Web frameworks and technologies*. Retrieved January 13, 2024, from https://survey.stackoverflow.co/2023/#section-most-popular-technologies-web-frameworks-and-technologies

Stack Overflow. (2023c, June 13). *Stack Overflow developer survey*. Retrieved January 13, 2024, from https://survey.stackoverflow.co

TechCrunch. (2010, May 18). *Google makes $68.2 million cash offer for global ip solutions*. Retrieved January 29, 2024, from https://techcrunch.com/2010/05/18/google-makes-68-2-million-cash-offer-for-global-ip-solutions/

The Captury GmbH. (2018, October 18). *Streaming library from CapturyLive*. Retrieved January 25, 2024, from https://github.com/thecaptury/RemoteCaptury

three.js. (2010, March 23). *Javascript 3d library*. Retrieved January 25, 2024, from https://github.com/mrdoob/three.js

Tone.js. (2014, March 11). *A web audio framework for making interactive music in the browser*. Retrieved January 25, 2024, from https://github.com/Tonejs/Tone.js

Tsang, A. (2014, July 29). *TJ Holowaychuk passes sponsorship of Express to StrongLoop*. Retrieved January 14, 2024, from https://web.archive.org/web/20161011091052/https://strongloop.com/strongblog/tj-holowaychuk-sponsorship-of-express/

van Rossum, G. (2008, August 11). *Why is Python a dynamic language and also a strongly typed language?* Retrieved January 15, 2024, from https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language

Volpe, G. (2003, April 22). *Computational models of expressive gesture in multimedia systems*. Retrieved April 4, 2023, from https://theses.eurasip.org/theses/157/computational-models-of-expressive-gesture-in/download/

W3Schools. (n.d.). *Javascript history*. Retrieved January 15, 2024, from https://www.w3schools.com/js/js_history.asp

Wallace, D., Watson, A., & Mccabe, T. (1996, 1996-08-01). Structured testing: A testing methodology using the cyclomatic complexity metric.

Web Audio API. (2021, June 17). Retrieved April 4, 2023, from https://www.w3.org/TR/webaudio/

WebRTC: Real-time communication in browsers. (2023, March 6). Retrieved April 3, 2023, from https://www.w3.org/TR/webrtc/

WebXR Device API. (2023, March 3). Retrieved April 4, 2023, from https://www.w3.org/TR/webxr/

Worrall, D. (2018). Sonification: A prehistory, 177–182. https://doi.org/10.21785/icad2018.019

ws. (2011, November 9). *Simple to use, blazing fast and thoroughly tested websocket client and server for node.js*. Retrieved January 25, 2024, from https://github.com/websockets/ws

Ye, M., Wang, X., Yang, R., Ren, L., & Pollefeys, M. (2011). Accurate 3D pose estimation from a single depth image. *2011 International Conference on Computer Vision*, 731–738. https://doi.org/10.1109/ICCV.2011.6126310

Yegulalp, S. (2015, September 10). *IBM snaps up StrongLoop to add Node.js smarts to BlueMix*. Retrieved January 14, 2024, from https://web.archive.org/web/20221013083101/https://www.infoworld.com/article/2982876/ibm-snaps-up-strongloop-to-add-nodejs-smarts-to-bluemix.html

Yoshino, T. (2015, December 7). *Compression extensions for WebSocket*. Retrieved January 25, 2024, from https://datatracker.ietf.org/doc/html/rfc7692

You, E. (2021, September 29). *Introduction: The progressive framework*. Retrieved January 14, 2024, from https://vuejs.org/guide/introduction.html#the-progressive-framework

Zhou, Q.-Y., Park, J., & Koltun, V. (2018). Open3D: A modern library for 3D data processing. *arXiv:1801.09847*.

µWebSockets. (2016a, March 21). *Expressive middleware for node.js using ES2017 async functions*. Retrieved January 25, 2024, from https://github.com/koajs/koa

µWebSockets. (2016b, March 21). *Simple, secure & standards compliant web server for the most demanding of applications*. Retrieved January 25, 2024, from https://github.com/uNetworking/uWebSockets

Appendix Listing

A. Appendix