RagRug: A Toolkit for Situated Analytics

Philipp Fleck[1], Aimée Sousa Calepso[2], Sebastian Hubenschmid[3], Michael Sedlmair[2], Dieter Schmalstieg[1]

[1]Graz University of Technology [2]University of Stuttgart [3]University of Konstanz

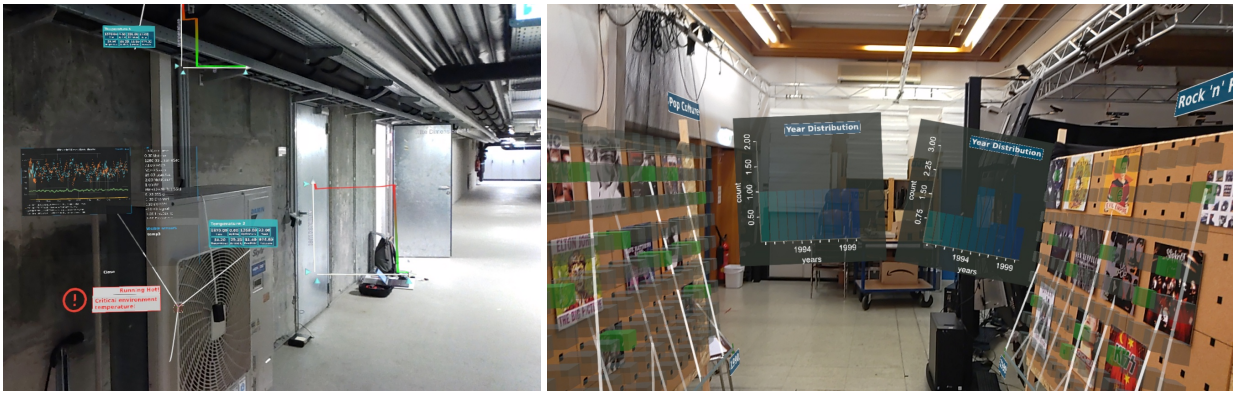

Fig. 1. Example application built with RagRug: (left) Hot air emissions in the basement. All measurements are taken in real-time by stationary sensors observing heat distribution. Multiple sensors are visually annotated with panels giving detailed sensor characteristics, including two types of time-series plots of temperature. (right) An application supporting the design of an exhibition of musical artefacts. Each of the two exhibition walls on the left and right hand side has been decorated with multiple exhibits. For each decorated position, a leader line extends to a timeline shown at the bottom of the wall.

Abstract—We present RagRug, an open-source toolkit for situated analytics. The abilities of RagRug go beyond previous immersive analytics toolkits by focusing on specific requirements emerging when using augmented reality (AR) rather than virtual reality. RagRug combines state of the art visual encoding capabilities with a comprehensive physical-virtual model, which lets application developers systematically describe the physical objects in the real world and their role in AR. We connect AR visualizations with data streams from the Internet of Things using distributed dataflow. To this end, we use reactive programming patterns so that visualizations become context-aware, i.e., they adapt to events coming in from the environment. The resulting authoring system is low-code; it emphasises describing the physical and the virtual world and the dataflow between the elements contained therein. We describe the technical design and implementation of RagRug, and report on five example applications illustrating the toolkit's abilities.

Index Terms—Augmented Reality, Visualization, Visual Analytics, Immersive Analytics, Situated Analytics

1 Introduction

For the past few years, one can observe increased interest in combining visualization (Figure 2, left) with new user interface technologies, such as virtual reality (VR) and augmented reality (AR). While *immersive visualization* (Figure 2, middle) is usually implemented in VR, *situated visualization* (Figure 2, right) is grounded in the real world and, therefore, implemented in AR. With a mobile AR display, the user is freed from having to bring the data to the workplace and, instead, may go where the data belongs.

In the notion of Willett et al. [72], "situated" means in perceptual proximity to a physical *referent*, i.e., a *meaningful* object close to the user. The requirement to have a meaningful referent is an important distinction between immersive visualization and situated visualization: For example, overlaying an X-ray visualization of hidden electrical cables on a wall is a situated visualization, because the wall is a meaningful referent for the cable visualization. An AR overlay showing a bar chart of sales figures on the same wall may be an immersive visualization, but *not* a situated visualization, because the wall has no semantic connection to the chart other than serving as its canvas. Referents are key for situated visualization. By extension, the same is true for *situated analytics*, i.e., analytic work supported by situated visualization. Situated analytics near referents is not only relevant for *passive referents* (plain objects) but even more for *active referents* which provide their own data, e.g., via the Internet of Things (IoT).

Despite its promise to grant access to just the right data, anytime and anywhere, few works explore situated analytics for lengthy or complex tasks, such as visualization authoring [14, 15] or data exploration [18]. In contrast to immersive analytics, which successfully capitalizes on providing unlimited virtual "space to think" [2, 4, 6], it appears that the benefits of situated analytics are harder to manifest.

What causes this discrepancy? Likely, an important cause is that *including physical referents into visualizations proves challenging*. Immersive analytics essentially provides a gigantic canvas, free of any referents. In contrast, situated analytics needs to carefully combine the physical and the virtual, and must respect that referents can also confound interactive visualization. It is not obvious how to support fluid interaction [17] or encourage analytic depth [44] when one must interleave task execution in the real world with data analysis performed in the virtual world. In fact, the few situated visualization scenarios that emphasize analytic depth [45, 64] avoid the aforementioned difficulties by using paper artifacts. Arguably, pieces of paper are "lesser" referents in the sense that they are neither unique, nor is the location of tasks

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

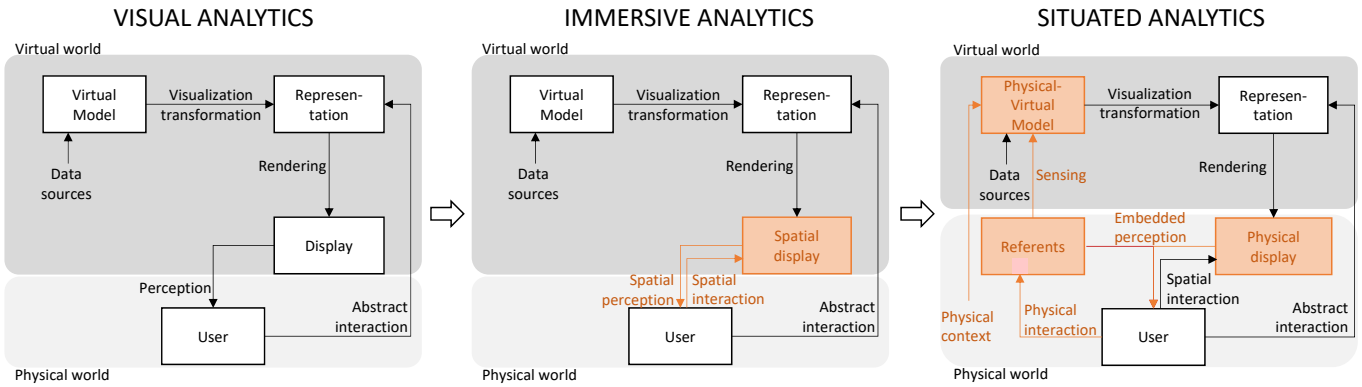

Fig. 2. (left) Desktop visualization/analytics has no notion of the physical world. (middle) Immersive visualization/analytics introduces the notion of a spatial display, which enhances perception and affords spatial interaction, such as ego-motion. However, the notion of a physical world is still absent. (right) Situated visualization/analytics explicitly introduces the real world in the form of physical referents and physical displays. One may manipulate the visualization directly via abstract interaction or indirectly via physical interaction with referents.

involving paper referents particularly relevant. We strongly believe that, with better software toolkits, building compelling situated analytics applications will become just as easy as building immersive analytics applications. At the moment, however, such toolkits for situated analytics are still missing.

In this paper, we attempt to close this gap by introducing *RagRug*, a toolkit for building situated analytics applications. The name RagRug was chosen to reflect the heterogeneous nature of AR. Our toolkit weaves "rags", i.e., virtual or physical components, into a "rug", i.e., a composite structure of high practical value. In summary, our work makes the following contributions:

- **Requirements**: We investigate the differences between immersive analytics and situated analytics, and we use this investigation to bring out the special requirements that situated analytics has over immersive analytics (Section 3).
- **RagRug toolkit**: We contribute the RagRug toolkit and describe how it fulfills the needs of situated analytics that have not been previously addressed by other toolkits. In particular, RagRug leverages a *representation of referents in an explicit physical-virtual model* and uses a *distributed visualization pipeline* (Section 4).
- **Application examples**: We present results covering four different use cases, ranging from spatially small to large environments and from primarily virtual to primarily physical data (Section 5).

2 Related Work

We briefly review the state of the art in situated analytics and the various domains to which it is related. Specifically, we compare toolkits for visualization with toolkits for VR, AR, and IoT. We also review authoring solutions for these domains.

2.1 Situated visualization and analytics

Thomas et al. [67] speak of situated analytics when a visualization is perceived close in space and time to its referent. This definition implies that we know the referents' spatiotemporal location. In the simplest form, one can only detect the presence of referents, not their exact spatial location [5, 78]. Yet, a lot of research in AR has focused on object detection and registration. For example, referents can be detected using Wifi [35] or ultra-wideband radio [33]. Most approaches use a detector based on some form of computer vision [35].

Given such precise registration information, situated visualization techniques can *embed* [72] visualizations into the perception of referents. Data that has an inherent spatial characteristic is an obvious candidate for embedded visualization, as simple overlay onto a referent at 1:1 scale already provides the user with additional insight. Such visualizations can reveal the correspondences between real and virtual dimensions, e.g., for temperature [26], viticulture [38], geological formation [40], water levels [68], corrosion [69], pollution levels [71], construction site progress [80], or CAD models [30].

There are also attempts at situated visualization of abstract data without an inherent spatial dimension, e.g., tourist maps [14], charts [15], nutritional information [18], bibliographies [45], or free-form annotations given as post-it collections [64]. However, these works do not expose their situated visualization techniques in a general-purpose toolkit, as we do.

2.2 Toolkits for visualization

Toolkits are essential components of visualization research, as they allow practitioners to build more *expressive* visualizations, and to do so with enhanced *ease of use*. Visualization toolkits, such as InfoVis Toolkit [21], ProtoVis [10], Prefuse [28], D3 [11], or Vega [60] brought data visualization to a broad audience of programmers, web designers, and other target groups.

Broadly speaking, contemporary Visualization toolkits, such as Vega-Lite [1], generate visualizations by applying a series of visual encoding transformations to the raw input data elements. The vocabulary of transformations is typically given as a grammar of graphics [9]. The resulting 2D visualizations are presented on desktop computers or embedded in web pages. Consequently, this class of visualization toolkits usually does not support 3D visualizations or interaction beyond mouse and keyboard.

Recent research has focused on toolkits for immersive analytics, which bring data visualization to VR. Moving to VR makes it necessary to generate a visual representation in 3D, even if the visual encoding of the visualization may still be restricted to 2D. Several toolkits for immersive analytics, such as DXR [62], IATK [16], and U2VIS [57] have been built with the Unity game engine, which is currently the most popular choice for VR game development. DXR uses a grammar of graphics to generate visualizations represented as graphical game objects, thereby establishing a workflow on a VR platform comparable to desktop visualization toolkits. IATK has similar visual encoding abilities as DXR, but much better performance on large data sets, since its visual encoding runs in geometry shaders on the GPU. MRAT [51] is an immersive analytics toolkit with a different purpose. It lets developers instrument AR applications by visual programming for data collection during user sessions. This collected data is then synthesized into 2D and 3D visualizations for further analysis. VRIA [13] takes an alternative implementation approach by relying on web technologies, in particular, the WebVR standard. While this approach benefits from its openness and avoids being tied to a particular platform, it lacks the rich community and ecosystem built around Unity.

A common limitation of all these immersive analytics toolkits is that, while they support AR displays to some extent, any support for referents (and, consequently, for situated analytics) is extremely limited.

In the words of Milgram and Kishino [50], they do not provide much help to increase the "extent of world knowledge".

2.3 Toolkits for VR, AR, and IoT

Besides delivering 3D graphics, one of the main requirements that VR/AR toolkits must address is the support for a wide variety of input and output modalities. There are many different display form factors, different tracking devices, and other aspects which are hardly considered in desktop applications. An even more diverse range of devices than required in VR/AR must be supported in IoT, and such devices increasingly play a pivotal role as active referents in AR applications.

IoT scenarios require a networking middleware capable of forwarding events and data streams in a flexible manner [59]. Several approaches combining AR and IoT use such a networking middleware to deliver situated visualizations (at least very simple ones) for electronic devices [52]. A popular scenario in this space is appliance remote control [8,24,35,42,61]. Another special case of AR remote control targets mobile robots [37] and drones [20, 79]. Other AR/IoT applications cover physically larger areas, such as traffic [54] or factories [3], or target composite devices, e.g., an audio mixer [46].

Of course, we are not the first to consider networking middleware for device communication in AR or related fields. Specialized tools for establishing a *distributed dataflow*, which let communicating devices act as sending and receiving nodes in a network, have been used in VR [66], AR [58], and robotics [56]. However, these approaches are rather narrow in their scope, hardware support, and protocols. RagRug aims to achieve broader coverage of a wide selection of IoT devices.

2.4 Visualization authoring

The coding aspects of immersive and situated visualization have been addressed with domain-specific languages [41,47] and visual programming [23]. This is sufficient if the visualizations have a known scope (for instance, user session analysis [51]), but a general-purpose authoring process must also consider content creation, i.e., dealing with geometry, appearance, animation, and so on. To some extent, both coding and content creation for AR can be simplified by simulating the real environment [19,55]. However, simulations are not always faithful enough and cannot be used if the environment is not known in advance. In these (frequent) cases, situated authoring – in the sense of authoring at the actual physical location – is the only option [49].

Meanwhile, immersive analytics toolkits [14–16,62] transfer some or all of the visualization pipeline specification (i.e., authoring) into the immersive domain. While this approach works well with VR, it does not include the physical world.

Jansen and Dragicevic [34] propose to address this discrepancy by leveraging *instrumental interaction* [7], a generalized form of direct manipulation, which distinguishes domain objects (in our case, the visualization data) from interaction instruments. Interaction instruments combine physical and virtual parts (see also Figure 2). Indeed, referents can serve as physical instruments in situated visualization, while the stages of the visualization pipeline can serve as virtual instruments [34]. Examples for physical instruments are proxemic interactions [42], which leverage the relative pose of user and referents as input, or touch interactions on ordinary surfaces via hand tracking [25, 27, 73, 74]. Because of its natural match with AR, instrumental interaction provides a powerful framework to perform situated authoring. Notable examples from the literature apply this form of authoring to embedded remote controls [73], creation of status visualizations [29,36], or visual programming of electronic devices [65].

RagRug supports a similar form of authoring. It enables the combination of physical and virtual instruments in a completely open manner, using free-form dataflow.

3 Requirements

The requirements of a situated analytics toolkit are diverse, since it needs to span visualization, VR, AR, and IoT. To give a concise overview of the requirements, we organize the discussion around four topics that a situated analytics toolkit must address in order to be of practical value, i.e., to avoid making unrealistic simplifying assumptions: context-awareness, physical-virtual models, reactive behaviors, and situated authoring.

3.1 Provide context-awareness for visualizations

In order to incorporate referents into our visualizations, a first and foremost objective is to **R1** *make visualizations context-aware* with respect to changes in the real world. Context is crucial to accommodate an AR user who cannot be expected to explicitly specify every minute detail of the current situation to the AR system while being immersed in a physical task. Context-awareness reduces the need for explicit specification. For example, the AR system may assume that commands issued by the user refer to the currently present referents, unless it is otherwise indicated.

Therefore, RagRug must connect visualizations to real-time events and data streams from sensors worn by the user or placed in the environment. The context acquired from these sensors can be diverse. For example, if a referent is moved, a camera tracks the movement, and an embedded visualization registered to the referent moves along with it. If the incident lighting changes (e.g., if the window blinds are opened), an illumination sensor fires, and the system changes colors and contrast of the see-through visualizations. If the user performs an invalid action (e.g., attaching the wrong machine part during a guided repair procedure), a monitoring component notices the mistake, and a warning is displayed. None of these events would likely be considered by a user as an explicit interaction with a visualization. Yet, from the point of view of the situated analytics application, they are regular events that must be processed by the application logic.

3.2 Support a comprehensive physical-virtual model

Referents do not only make themselves known through short-lived events, they also have persistent or semi-persistent characteristics, such as shape, location, or identity. Situated analytics applications must know about these characteristics. For maximum flexibility, a data-driven application should not contain any hard-coded data about referents. Instead, it should query a suitable database to obtain relevant data about references dynamically. We therefore require that RagRug must **R2** *support the developer in creating and maintaining a comprehensive physical-virtual model*. The model can be understood as a kind of digital twin. It comprises not only virtual data from arbitrary external sources, but also the most recent state of the referents and other characteristics of the physical world. Thus, the model is essential in correctly linking visualizations to referents.

Despite the long standing recognition [63] of a need to connect AR to data about the real-world, we are not aware of any toolkit-level support for the inclusion of a general-purpose physical-virtual model in the visualization authoring process. We require means to define the model, store it, fill it with data efficiently (i.e., in an automatic or at least a computer-assisted manner), and retrieve the data from the model in a flexible way (i.e., using an appropriate query mechanism).

3.3 Make visualizations reactive

The data sources implied in these examples are varied. Nonetheless, building a flexible interface for data sources *known in advance* is relatively simple. In contrast, building an interface for communication that *adapts to new data sources on the fly* is significantly harder. This kind of requirement is not addressed well by current immersive analytics software, which assumes that all relevant objects – in our case, the referents – are determined at startup and do not change while interacting with the visualizations (a property which Lacoche et al. [39] call "auto-configurable").

RagRug strives to be fully context-aware and not just auto-configurable. Therefore, a further objective is to **R3** *make the visualization pipeline reactive*, i.e., a re-evaluation of the pipeline must be triggered after every external change. For example, marks in a visualization may be changed whenever a device is powered on or off.

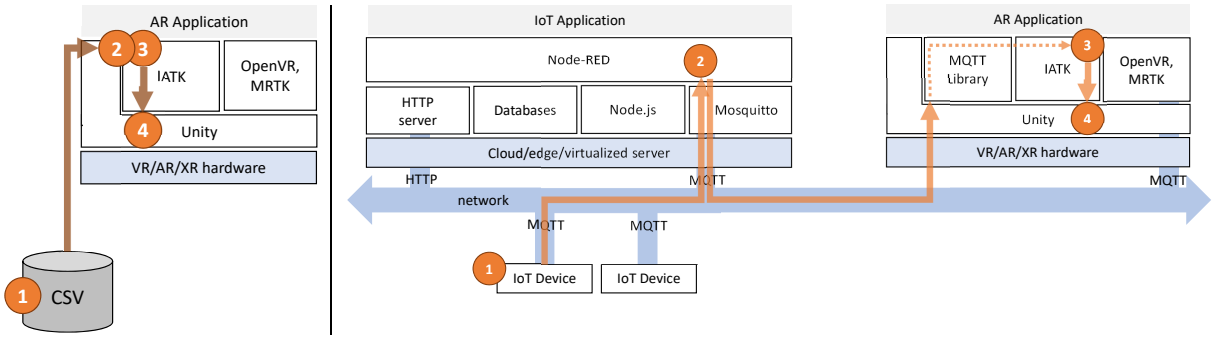

Fig. 3. (left) The visualization pipeline of a standard IATK application consists of (1) data acquisition from a CSV file, (2) data filtering and (3) visual encoding in IATK, and (4) rendering in Unity. (right) Combining an IoT application environment with IATK yields a distributed visualization pipeline: (1) Data is acquired using MQTT; (2) data is filtered using dataflow in Node-RED; (3) data is visually encoded in IATK; (4) visualization marks are rendered in Unity. However, this approach introduces a gap between IoT and IATK (dashed orange arrow) and consequently lacks a unified programming model.

3.4 Enable situated authoring of visualizations

A physical-virtual experience with numerous, diverse elements can be difficult to oversee. Conventional development cycles of modifying code, then testing it, tend to become arduous when coding requires a desktop computer, but testing requires physical interaction with referents [23]. A lower entrance barrier and better expressive leverage [53] can be achieved by allowing *situated authoring at runtime* [48].

To address this requirement, we propose to combine the aforementioned reactive visualization pipeline with runtime interpretation of application logic. Combining these two technologies enables the visualization designer to modify every aspect of the system either by scripting or visual programming, without ever having to shut down the system. Designers can interleave coding, data modeling, and spatial interaction for rapid design iterations, and experimenters can deal with unforeseen situations by adjusting code and content on the fly as needed. RagRug must therefore progress from "authoring of situated visualizations" to **R4** *situated authoring of visualizations*. What is required for this kind of authoring?

Obviously, RagRug cannot solve all needs of situated authoring with a fixed set of interaction tools. Hence, instead of including interaction tools directly, it should offer a programming model for creating novel instrumental interaction tools [34]. We will explain below how RagRug achieves this requirement by providing a uniform programming model for all components (IoT services, AR displays, databases, etc.), thereby insulating the developer from the difficulties of dealing with diverse and complex components.

Item	Description
R0	Provide 3D visual encoding
R1	Make visualizations context-aware
R2	Support a comprehensive physical-virtual model
R3	Make the visualization pipeline reactive
R4	Support situated authoring of visualizations

Table 1. Five requirements guide the design of RagRug

4 Toolkit Design

The visualization capabilities of RagRug build on the state of the art in immersive analytics. In particular, RagRug builds on IATK (Figure 3, left), which has best-of-class abilities with respect to immersive analytics. IATK is fast, its code is well-maintained, and its integration into Unity lets developers benefit from the rich VR/AR ecosystem evolving around Unity. This motivated us to adopt IATK. Consequently, RagRug inherits all visual encoding abilities of IATK and produces the same visual representations. This capability addresses the implicit requirement **R0** of *providing 3D visual encoding capabilities*, which we add to the list of requirements (Table 1).

The added value of RagRug comes from its substantially extended *interaction capabilities* when compared to IATK or other toolkits in its class. The term "interaction capabilities" is used in a broad sense here to mean explicit input provided by the user (e.g., via 3D controllers or touch surfaces), but also any interaction with the environment. It has been argued that interaction is the perpetual step-child of visualization [75]. We feel that progress in the new field of situated analytics critically depends on promoting interaction in visualization with enhanced interaction capabilities. Requirements **R1**-**R4** all concern aspects of the interaction with the environment and are not (or not sufficiently) addressed by IATK, which focuses only on visual encoding (**R0**).

First, IATK lacks any capabilities for context-awareness. It has no support for streaming data from sensors or receiving events from external sources. Even if external sources are added (e.g., via additional Unity libraries), the static data model underlying IATK is ill-suited for the dynamic event streams implied by **R1**.

Second, IATK procedurally generates visual marks from a *static* dataset. It is possible to embed referent representations in such a dataset. However, that does not make such referents first-class citizens. Updating the physical-virtual model (containing the referent description) requires manually replacing the dataset, which is hardly what is implied by **R2**.

Third, IATK offers only a finite set of built-in interactions (brushing of data points, axis scaling, etc.). This limitation is not so much a result of its focus on immersive analytics, but rather a consequence of its lineage from Vega-Lite, which favors ease of use over expressiveness when it comes to reactive behaviors. The precursor of Vega-Lite, *Reactive Vega* [60], explicitly made input streams and events first-class citizens via its *event-driven functional reactive programming* model (E-FRP) [70]. FlowMatic [77] has recently demonstrated how E-FRP can be applied to authoring 3D interactions. To address **R3**, we use a similar extension inspired by E-FRP in RagRug.

Fourth, IATK has only limited support for situated authoring. One can use its desktop interface for limited visual programming, but this solution does not extend to provisioning new context-aware data sources or re-coding the reactive behaviors of the visualization to deal with such sources. Full support for **R4** demands that not only the sources and sinks (effects on the visualization) can be re-coded on the fly, but also all transformations between source and sink.

The above appraisal of IATK reveals which capabilities are lacking. In the remainder of this section, we will systematically expand our software architecture to address them, while carefully ensuring that existing benefits provided by each of the components are retained.

4.1 Context-awareness

Context-awareness (**R1**) in RagRug is provided by adopting IoT software. We will therefore begin our discussion by describing the state of the art in this area. IoT devices, such as sensors or actuators, are typically headless, i.e., they can only communicate over the network. Since IoT devices are also embedded systems without programmability (their purpose is only to send data or receive commands), the actual IoT application logic is executed on a cloud (or edge) server. Since

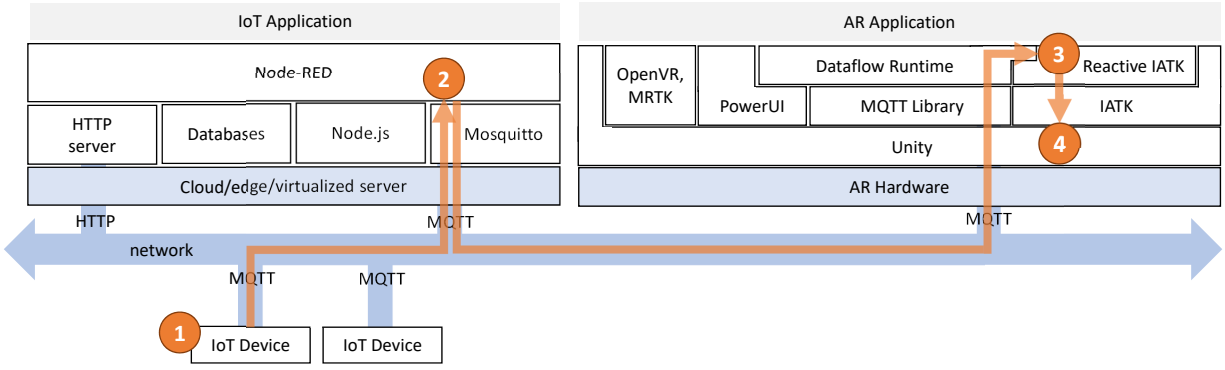

Fig. 4. The RagRug client extends the standard IATK application model (shown in Fig. 3) with third-party components (MQTT, PowerUI) and custom components (reactive IATK, dataflow runtime). As a result, a unified programming model based on dataflow and reactive programming is now available throughout the entire system. The client-side visualization pipeline (orange arrow) does not differ in programming style from the hub-side visualization pipeline.

the server is not as resource-restricted as the IoT devices are, it conveniently leverages a standard multi-tier web architecture, consisting of an HTTP server, a Node.js processor for Javascript applications, and database backends.

One of the most widespread application frameworks for IoT applications is Node-RED[1]. It exposes a dataflow programming model, which supports application logic built by connecting nodes into a pipes-and-filters [12] graph, with data streaming through the graph. The operations assigned to a node can be chosen from a standard library (e.g., nodes for averaging values or for merging two streams) or written as custom Javascript functions. Authoring for such an application framework is frequently done in a low-code manner via a visual programming tool, which lets the user interactively build a node-link diagram representing the dataflow graph, intertwined with writing small amounts of Javascript code for customizing the nodes.

If Node-RED developers follow the recommended practice of keeping their Javascript code free of side-effects, the result conforms to E-FRP [70]: Events (in our case, from IoT sensors or other real-time sources) are assembled into streams consisting of time-stamped event sequences. Every new event received by a source node triggers an update to its dependent nodes in the dataflow graph, until the dataflow arrives at the sinks.

The dataflows in Node-RED are computed locally in a Node.js environment. Since the need for distributing dataflow across a network frequently arises, Node-RED is typically combined with *publish-subscribe* (pub-sub) communication [12], e.g., by leveraging the *message queue telemetry transport* (MQTT) protocol. MQTT is a lightweight network protocol, which is designed to efficiently distribute device data across a network [31]. Publishers tag their messages with a *topic* upon sending them, and subscribers can express interest in various topics. A broker (we use the open-source tool Eclipse Mosquitto[2]) is responsible for routing the messages to the relevant subscribers. Publishers and subscribers can either be nodes of a dataflow graph executed in Node-RED, or, arbitrary hosts on the network. Pub-sub can connect nodes that know of a shared topic, but do not know each other directly (a concept also known as *referential transparency*). The use of topics significantly eases the implementation of context-awareness. With a topic agreed in advance, publishers (e.g., sensors) and subscribers (e.g., visualization applications) find each other at runtime without any additional directory services.

Since it also represents a known point of contact for a particular environment, we call the ensemble of IoT services built around Node-RED and Mosquitto a *hub*.

[1] https://nodered.org
[2] https://mosquitto.org

4.2 Physical-virtual model

To provision a physical-virtual model (**R2**), we leverage the hub's web-centric architecture, in particular, the database backend. In industrial environments, such databases may, for instance, be used in production planning or to log sensor readings for documentation. In RagRug, we utilize the databases to store the physical-virtual model. For example, we store sensor readings as time series in InfluxDB[3], CAD or 3D scan data describing the physical environment in CouchDB[4], and relational data, such as room codes, in Postgres[5].

The main advantage of having such a database backend does not come from the creation of the data, which, in most cases, must still be done manually, either by manual entry or by manually collecting and converting the required data sources. Rather, the benefit of the database backend comes from the ability to answer queries issued by a client. Rather than having to push a particular dataset manually to the client, the dataset can be deposited in the database, and the client can query it dynamically. For example, upon entering a room, the client can issue a query about all data elements tagged with the current room code, and receive an up-to-date representation of all referents that have been stored in the database. Hence, database queries do not only make it easy to organize the physical-virtual model, they also provide a degree of referential transparency for the static aspects of a situated analytics application.

4.3 Reactive programming

Up to this point, we have described two standalone platforms, one for immersive analytics – e.g., an AR client running Unity, Microsoft Mixed Reality Toolkit (MRTK), and IATK – and one for IoT applications (a hub running Node-RED, Mosquitto and other services). Together, the two platforms offer a lot of functionality in a tried and tested form. If we want to use this functionality in a combined application, it makes sense to first consider a lightweight integration that lets developers tap into both software ecosystems as needed.

Indeed, establishing a rudimentary connection between the two platforms is rather simple (Figure 3, right). We only need to add an MQTT library to the client and connect it to dataflow sinks in Node-RED in order to receive event streams. Then, we write code to change the IATK visualization whenever new events arrive. The result is a split visualization pipeline, where data acquisition is accomplished by the IoT devices; data preprocessing is performed in Node-RED; visual encoding is done by IATK, and rendering is carried out by Unity.

However, this approach suffers an important drawback. It has good expressiveness (all important things can be done), but rather mediocre ease of use: The two platforms have noticeable different programming models, and building complex, scalable applications is cumbersome. In

[3] https://www.influxdata.com
[4] https://couchdb.apache.org
[5] https://www.postgresql.org

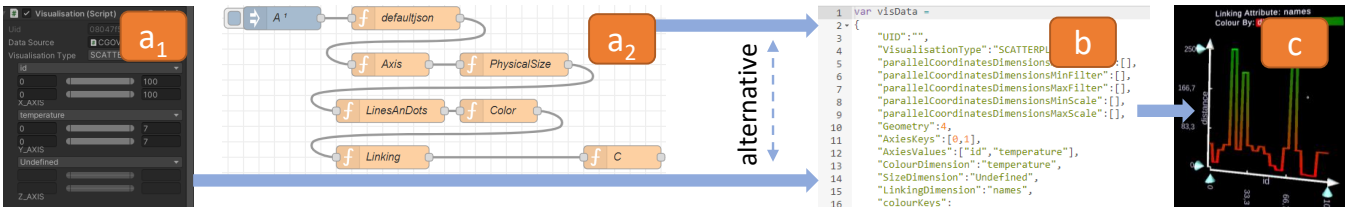

Fig. 5. Comparison of authoring in IATK and RagRug: (a_1) traditional IATK specification in the Unity editor; (a_2) an equivalent IATK specification in the Node-RED editor, (b) the IATK-internal JSON representation created by either a_1 or a_2; (c) the resulting visualization.

particular, writing custom IATK code for every kind of incoming event sequence is inelegant and prone to errors. We would much prefer if IATK can naturally participate in the dataflow, ideally, using the same reactive programming style as applied in the rest of the system.

Making IATK reactive Hence, we must extend IATK to support reactive programming (**R3**). In other words, the visualizations of IATK must accept updates outside of the limited built-in interactions. To understand our solution to this issue, we have to take a closer look at the internal operation of IATK. Recall that a grammar of graphics defines production rules that can be applied on data sources to describe the visual encoding process in a flexible manner. In the case of IATK, the productions rely on a decorator pattern [22] to chain design commands to a basic visualization object. The basic visualization object is, in turn, linked to a data provider object, which is a container object representing a set of data points in memory. Internally, IATK stores data points in arrays on the GPU, where a geometry shader converts data points on the fly into 3D geometry for visualization marks.

This architecture implies that a visualization may be changed in two ways: First, the values that make up the data points (or the number of data points) may change. In this case, it suffices to re-transmit the changed or new data points from CPU to GPU, and the visualization will automatically be updated by the geometry shader. Second, the definition of the marks may change. In this case, IATK visualization structures (represented internally as C# objects) must be updated, and we want to carry out this update in a minimally invasive way.

Hence, we trigger both types of change operations from the external dataflow. The first case, updating data points, is addressed with an MQTT event handler which updates the data provider (i.e., the CPU copy of the data point array) and triggers a synchronization from CPU to GPU. As a result, an updated visual representation will be generated when the geometry shader is invoked the next time.

The second case, updating visualization structures, repurposes the internal mechanism of IATK for loading and saving a visualization as a JSON string. Loading relies on lazy evaluation (i.e., only those parts present in the JSON string are updated), which perfectly fits our requirements of applying incremental changes at runtime rather than building a visualization anew from scratch. We expose the IATK load function to the handler for incoming MQTT events. It creates a JSON string filled with the desired parameters, and passes it to the load function of IATK. By passing JSON, our Javascript code does not have to know about the internal C# representation of visualizations in IATK. This advantage comes at the cost of having to encode and decode a JSON string and let IATK decode it again. However, the strings are short, and we have not observed any slowdowns using this approach. Another advantage is that any third-party extensions to IATK will inherently continue supporting the load functionality, so we can be confident that our update method will be maintenance-free in future versions of IATK.

Client-side dataflow The reactive programming extension to IATK as presented above has an important limitation: Visualizations directly depend on events delivered by the dataflow. Since the dataflow is still restricted to operate within Node-RED, the client can only subscribe to MQTT messages and apply the message content to IATK. It cannot run any dataflow *locally* to further process events. Instead, any interactive or reactive behavior must be handled via a round-trip to Node-RED on the hub. This behavior would also apply to 3D interaction emerging from local 3D input on the client (e.g., 3D hand tracking events delivered by MRTK). Sending data from the client to the hub and back after processing unnecessarily increases complexity and may induce unwanted latency. Thus, we desire the ability to establish a local dataflow on the client.

Since embedding another instance of Node-RED inside Unity for running client-side dataflow is not feasible, we built a lightweight runtime interpreter for dataflow (Figure 4). Sources of the dataflow can be events local to the client (e.g., from MRTK) or MQTT events received over the network. Dataflow sinks can be connected to IATK in order to make updates to visualizations, although other Unity game objects could serve as sinks as well.

Since we want all communication facilities to be uniform, we also allow local pub-sub inside the same Unity instance. Instead of routing messages through the hub, local subscribers receive messages directly with minimal latency. If desired, a copy of the message can still be distributed to the hub (e.g., for bookkeeping) and relayed to other hosts.

4.4 Situated authoring

To support situated authoring, we must strive for a unified programming model encompassing all components of RagRug. In the following, we describe how we enable Node-RED-compatible dataflow programming in Javascript (and therefore a distributed form of E-FRP) inside our AR client, which conventionally uses imperative C# as its native programming language.

With PowerUI[6], a Javascript extension to Unity, all application code can be written solely in Javascript on both hub and client. PowerUI has the ability to automatically expose the C# libraries we are using (such as the IATK, MRTK, and MQTT libraries) to Javascript. The Javascript code can be dynamically loaded and compiled just in time, which is important for agile development. If only the Javascript portion changes, the compiled Unity code does not need to be updated at all.

Our dataflow runtime for the client is written in Javascript as well. It reads unmodified dataflow graphs of Node-RED. During initialization of a RagRug application, Javascript code is automatically deployed from the hub to the client. This Javascript code contains the runtime library for executing the client-side dataflow, the exported dataflow graph exported for the client (in JSON format), and any Javascript functions for customizing the nodes.

The dataflow authoring in the Node-RED editor also integrates the grammar of graphics used by IATK. Rather than expressing a visual encoding as a series of decorator function calls in C#, the designer relies on a series of equivalent JSON strings wrapped in linked nodes of the dataflow. In that way, all steps in the visualization pipeline (and not only the visual encoding, as in original IATK) can be uniformly expressed as dataflow (see Figure 5).

A distributed dataflow is authored in the Node-RED editor in the same manner as a local one, with the only additional requirement that the dataflow must be appropriately split for deployment across multiple hosts in the network. From an authoring point of view, all other aspects remain unchanged. The only exception is that the author must bear in mind that some features may have restricted availability. For instance, 3D input events may only be available on an AR client.

In summary, we have found that uniform support of Node-RED-style dataflow with custom Javascript behaviors across the entire distributed

[6] https://powerui.kulestar.com

Aspect	IoT app.	IATK app.	RagRug
Language	**Javascript**	C#	**Javascript**
Visual encoding	no	**Vega-Lite**	**Vega-Lite**
3D rendering	no	**Unity**	**Unity**
Interaction	**reactive**	fixed	**reactive**
Passive referents	**via database**	no	**via database**
Active referents	**via MQTT**	no	**via MQTT**

Table 2. RagRug combines the advantages of IoT applications with immersive analytics capabilities in a unified reactive programming model.

system is not only easy to learn and use, but it also facilitates situated authoring very well, because it is easy to modify only a small portion of the distributed system. The changed Javascript code can be re-loaded and re-started at any time, even while the applications keeps running (for instance, while wearing a VR or AR headset). We have found that this automatic code deployment makes it much easier to work with mobiles or wearables. RagRug developers often spend their workday using a hybrid interface by wearing AR glasses while seated at their desktop computer. We find this behavior to be a serious demonstration of situated authoring (**R4**) in real life.

From the visualization developer's perspective, the combination of Javascript with two relatively minor (when compared to overall software complexity of the platform) extensions – reactive IATK and Unity dataflow – makes a significant difference, as it results in a unified programming model, as summarized in Table 2:

1. Javascript is the sole implementation language. (C# is still available if desired, but we have found no need for it.)
2. Most interaction processing can be set up as dataflow, which is easy to learn and yields modular, reusable code.
3. Authoring can be entirely conducted in the low-code evnironment of Node-RED.

Note that, as a byproduct, multi-user support can be achieved without any additional multi-user network library. For instance, one could forward all 3D events generated by a first AR client via MQTT to a second AR client, where it is made available alongside the local 3D events provided by the second AR client. A client application would be agnostic as to whether 3D events were generated locally or remotely. We describe another multi-user application in Section 5.3.

4.5 A simple authoring example: smart fridge

As a practical example to demonstrate authoring with RagRug, we build a *smart fridge*. We install a thermometer and a distance sensor inside the fridge (Figure 6). The thermometer data stream can be used to visualize if a proper temperature is maintained inside the fridge. If the temperature rises to exceed a critical mark, we use the distance sensor to check if the fridge door has been inadvertently left open and display an audio-visual alarm. We demonstrate two variants of the smart fridge application. In both cases, it is assumed that the distance meter and the thermometer are connected to local Wifi and publish their measurements via MQTT.

Variant 1 connects the AR client directly to the sensors without involving the hub. The dataflow (purely in the client) for Variant 1 consists of 12 nodes, as shown in Figure 7. The code for all nodes together amounts to about 60 lines of Javascript and another 40 lines of JSON for initialization for the IATK visualization. The code contains no complex control structures; most of it is boilerplate for initialization and for simple data transformations.

Variant 2 implements the same features, but demonstrates a distributed dataflow instead of pure client-side dataflow. The original node (labeled "L" in Figure 7) which displays an alarm if the door has been left open is split into two: The code for checking is executed in Node-RED, while the alarm display remains at the client. The two nodes are connected by MQTT messages: The hub node publishes "alarm" events, which the client node subscribes to (Figure 8). This version is preferred if scalability is important, for instance, if many sensors must be monitored.

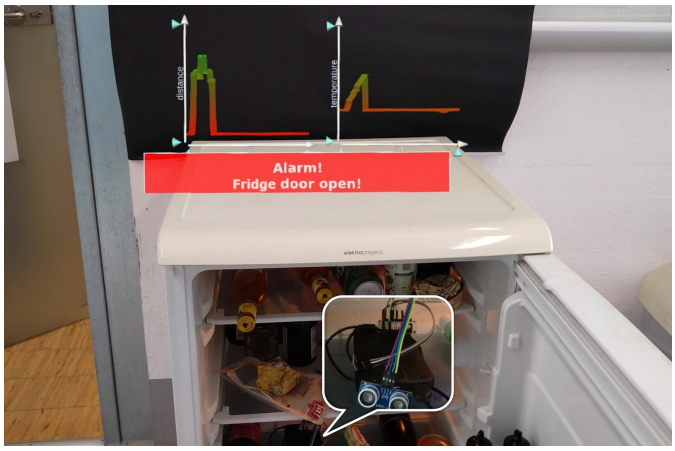

Fig. 6. The smart fridge displays line chart visualizations of its sensors and rings an alarm if the door is left open. The inset shows the sensors mounted inside the fridge.

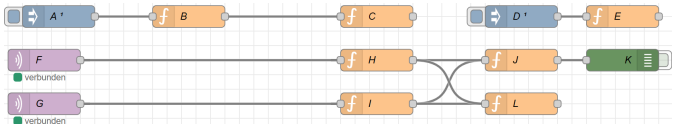

Fig. 7. Dataflow of the smart fridge example. The graph is a snapshot directly taken from the visual programming front-end of Node-RED: (A) trigger dataflow after 5 s delay, (B) send JSON template for the IATK visualization, (C) create IATK visualization, (D) trigger dataflow after 5 s delay, (E) move visualization into heads-up display, add collider so user can reposition the visualization, (F) subscribe to MQTT topic "temperature", (G) subscribe to MQTT topic "distance", (H) parse the temperature data and inject into dataflow, (I) parse the distance data and inject into dataflow, (J) insert data into IATK, (K) terminal debug output, (L) alarm if temperature $>10°$ C and distance >35 cm.

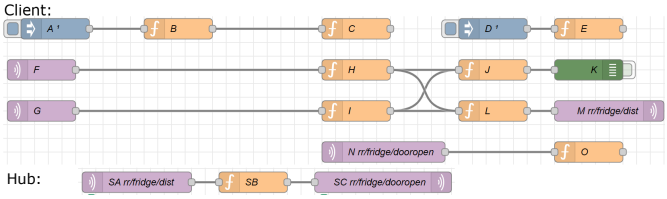

Fig. 8. Variant 2 of the smart fridge: (top) We replaced node (L) from Variant 1 (Figure 7) with the nodes (L)-(O). The new node (L) only reads the distance value and publishes it via node (M). Node (N) subscribes to the alarm topic and plays a sound via node (O). (bottom): The new Node-RED part of the dataflow checks if the distance value exceeds the threshold and publishes an alarm notification. Node (SA) subscribes to distance values and passes them to node (SB), which tests for the 35 cm threshold. If the threshold is exceeded, node (SC) publishes an alarm.

We will now describe how to author the fridge example. RagRug applications are strongly data-driven and typically involve only a small amount of procedural logic. Consequently, authoring largely consists of preparing the physical environment, recording the corresponding data items in the physical-virtual model, expressing the dataflow between items, and bootstrapping.

Spatial anchors To deliver spatially registered visualizations, RagRug requires a spatial model of the target environment. The spatial model must be prepared once in advance, either by mounting a target for image detection or scanning the room to create a "spatial anchor", i.e., a map of the immediate physical environment required by the on-board 3D tracking system of the AR device for self-localization. We generate a spatial anchor of the fridge room with the HoloLens and store it, associated with the room identifier. Anchors are stored as

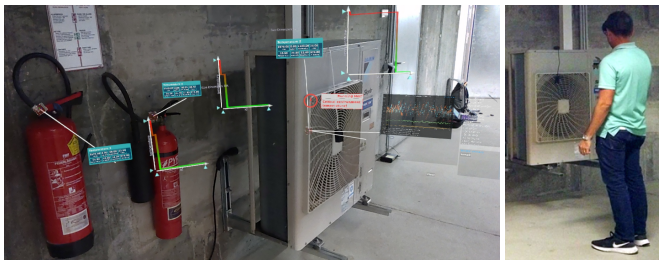

Fig. 9. (left) AR view of the automatically generated visualization, consisting of an info-panel on the left, with a leader line to the sensor, and a time-series graph on the right. (right) A user wearing a HoloLens looking at an instrumented ventilator.

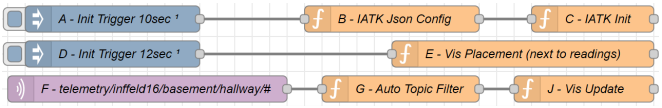

Fig. 10. Dataflow for one sensor in the basement: (A) trigger dataflow after 10 s, (B) load JSON visualization template, (C) spawn IATK visualization from JSON template, (D) trigger dataflow after 12 s, (E) place visualization next to sensor as depicted in Figure 9(middle), (F) subscribe to all MQTT messages on sensors in the basement, (G) filter messages based on current sensor, (J) update visualization with new data.

opaque binary objects (only the AR tracking system understands them). The locations of referents, visualizations, and other relevant objects are expressed in the coordinate system established by the anchor.

Physical-virtual model The fridge has only two sensors and one visualization. After physically mounting the sensors, we create JSON records for the sensors and store them in CouchDB. Finally, we create a visualization in IATK and store it in CouchDB as well. Visualizations are expressed by specifying the grammar of graphics understood by IATK. The visualization is annotated with its desired position above the fridge in the coordinate system of the anchor. Sensor locations are not required in this example.

Dataflow In the reactive programming model of RagRug, application logic is mostly expressed as Node-RED-compatible dataflow. Some of the dataflow nodes may include custom-written Javascript functions. We create an empty dataflow in the Node-RED editor and fill it with the nodes shown in Figure 7 and 8. Source nodes must be parameterized with the appropriate MQTT topic (in our case, corresponding to the sensors), and the client sink must be parameterized with the visualization template or contain custom Javascript code snippets (in our case, the condition to trigger the alarm).

Bootstrapping Once all information that makes up the physical-virtual model is stored in the databases of the hub (see above), bootstrapping is ready. When an AR client is started, only a stub procedure is loaded. The stub on the AR client connects to the hub listening on a known address on the wireless network. From the hub, the client retrieves the application code and all data items describing the environment. Retrieval of the data items (e.g., referents) is context-aware; for instance, the data items can be indexed by the current room identifier. In this way, application code is only loosely coupled to specific referents or other resources. This bootstrapping mechanism is applied for all use cases presented in this paper.

5 Case Studies

To evaluate RagRug, we used it to build four more sophisticated application examples. We selected scenarios across the spectrum of immersive and situated analytics, some new and some recreations of work from the literature. Our examples focus on context-aware handling of IoT (Section 5.1), large numbers of passive referents (Section 5.2), collaborative immersive analytics (Section 5.3), and fine-grained visualization embedding (Section 5.4). Below, we describe these applications as well as their implementations.

5.1 Basement: automatic visualization of IoT sensors

Scenario Consider a typical basement with heating, air condition, and power metering devices. A facility manager must regularly inspect the basement to investigate malfunctions and perform repairs. With an increasing number of devices and IoT sensors installed in the basement, the facility manager would like to know which devices are nearby, what their types and capabilities are, what the devices are currently doing, and what they have been doing in the past.

Device information and visualizations of the sensor data and other event streams over time should be shown as AR visualizations, spatially registered to the referents (i.e., the devices), as shown in Figure 9. As a prerequisite, we only assume that an edge server running the RagRug hub is installed and connected to the local network. Previous work investigating a similar scenario [55] relied on tedious manual authoring of such visualizations. In this example, we demonstrate how *IoT visualizations are generated automatically* from existing infrastructure and its documentation.

Physical-virtual model The model needs to store the location, type, and capabilities of the sensors deployed in the basement. Knowing about the sensors lets the AR client establish direct connections to the sensor and receive their live data streams. Hence, we require the spatial location of all the sensors, which is stored, alongside with the other sensor characteristics, in a relational database. Knowing about the *sensor location* enables the AR system to present information relating to a particular sensor as an embedded (i.e., spatially registered) visualization. If available, we are also able to parse a device's *geometric description* for parts, so that we can link a visualization to a specific part of a device. Geometric descriptions can be processed from a variety of common CAD formats (such as STL), which are also stored as part of the model. The sensor's location in the basement must be indicated manually, once upon installation, unless it is already known from a CAD model. For this purpose, we provide a simple pointing tool running on the AR device.

Sensor attributes other than location can often be derived from knowing the exact sensor type (e.g., in the form of a unique product number). Such type information can be used to automatically search a detailed device specification in a product database, encompassing the device's *parts*, its *capabilities*, and possibly the aforementioned geometric descriptions. Since no wide-spread standards for such product information exist yet, we have created our own database for the sensors used in our test setup. Most attributes characterizing a device are just strings (e.g., manufacturer and type) or numbers (e.g., voltage or update rate).

We create *visualization templates* in JSON for each sensor type. Particularly relevant are time-series visualizations of measurements (and, possibly, of events, such as malfunctions reported by devices) on a timeline. The JSON descriptions are downloaded by an AR client and passed to IATK to generate an instance of a visualization corresponding to a concrete sensor. We also prepare the application dataflow which is described in more detail below.

Implementation The basement application has two main parts. The hub part is responsible for autonomously filling the model with sensor information, which is revealed to any AR clients upon connection. The AR clients parse the sensor information, connect to the sensors, and display visualizations generated from their data.

Initially, the sensor part of the physical-virtual model is empty. Upon first installation of a physical sensor, a standard behavior of the sensor is to connect to a local wireless network and make contact on a standard service port, then start streaming data and wait for commands. We

Fig. 11. MoPop exibition wall with the timeline at the bottom and the year histogram to the right of the wall.

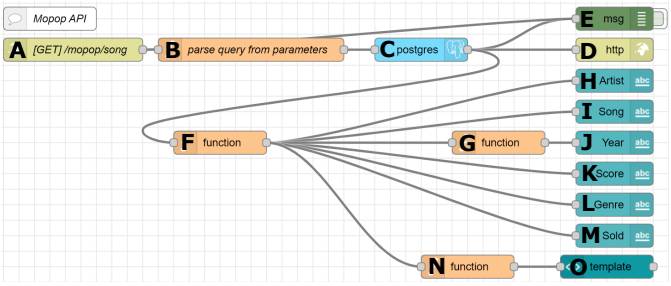

Fig. 12. Provisioning of musical meta-data in Node-RED is done via HTTP/REST (HoloLens) or via WebSockets (smartphone), since the meta-data does not fit into a single MQTT message: (A) receive a request for musical recording, (B) construct SQL query matching request, (C) run SQL query, (D) reply meta-data to AR client, (E) debug output, (F) multi-plexing of dataflow, (G) extract year from time/date string, (H-M) individual meta-data attributes sent to smartphone, (N) construct URL to download cover art image corresponding to musical recording id, (O) construct HTML image tag to display the cover art image.

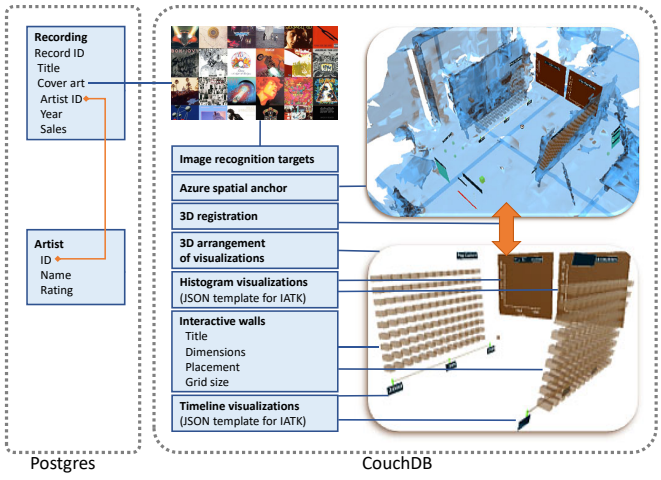

Fig. 13. The physical-virtual model of MoPop is stored in part in CouchDB (for plain text files and binary files, such as images) and in part in PostgreSQL (for relational data scraped from MusicBrainz).

assume that sensors use MQTT, although gateways for other communication standards could be added. A Node-RED application on the hub recognizes a new device and registers it in a database.

If the device messages contain type information, and a product database is available, the hub can automatically create a database record describing the newly connected sensor. We envision that such automatic creation of database records may be extremely valuable in large infrastructures with thousands of sensors, such as industrial facilities. In such a large-scale use case, setting up the necessary product databases may be well justified. If no product database is available, records for small sensor collections can be created manually.

When the facility manager deploys the AR client, it connects to the hub and downloads the relevant application data. From the sensor records, the client identifies the MQTT topics to which active sensors broadcast their data streams, and sets up a dataflow that visualizes the device data via the corresponding visualization templates. For each new sensor, a separate client-side dataflow is instantiated (Figure 10), which includes the possibility for custom filtering and visualization.

Since sensors send each data item only once, the hub automatically logs all sensor data in InfluxDB. Making the log available gives the client a choice to visualize the incoming data stream in real time or, alternatively, query the database for historic data.

To avoid clutter, the client interface only shows the visualizations for the sensors close to the user. The proxemic events of sensors entering and exiting the area of interest are conveniently expressed in the dataflow by evaluating a simple distance threshold. A heads-up display highlights the sensors currently in the field of view and can also be used to retrieve a full list of all sensors. Upon moving closer to a sensor, the semantic level of detail is switched from a compact glyph indicating sensor type to a full visualization, which can break up into multiple visualization panels arranged radially around the physical sensor (Figure 1, left). Panels can be repositioned manually as needed.

5.2 MoPop: computer-supported exhibition curation

Scenario Inspired by the Museum of Pop Culture, *MoPop*[7], in Seattle (USA), we created an application which simulates the work of a curator designing an exhibition room showcasing the evolution of two music genres. This example interleaves analytic work with physical manipulation of a large number of passive referents.

The curator must design a presentation of musical works (sleeves of musical recordings and promotional posters) on two opposing exhibition walls (one for pop, and one for rock music). The two walls display parallel timelines, starting in 1990 for the wall sides closer to the entrance and ending in 2000 close to the exit. To fill the walls, the curator picks exhibits and places them at a desired location on either wall (Figure 1, right). The curator wants the walls to be densely occupied with exhibits. Additionally, the most iconic works of the period should be presented, which have high sales figures and reputation. To further inform the overall design process, a visual guidance component provides feedback on the influence a particular choice of exhibit placement has on the desirable qualities of the exhibition..

Physical-virtual model MoPop stores its physical-virtual model (Figure 13) mostly in CouchDB. It contains a 3D floorplan of the exhibition space, where the locations of the physical objects, such as the walls, as well as the location of the various visualizations, have been added. For the configuration of the exhibition wall, we store various parameters, such as sizes and storage capacities of exhibition walls, display names, timeline parameters, 3D object names within Unity assets bundles, music data, images of record sleeves used as tracking targets, and so on.

A relational database stores a table of tracking targets, exhibit keys, and attribute keys to cross-reference them. Upon picking up an exhibit, the attributes are retrieved on the fly via a database query (Figure 12), so the collection could be extended at runtime.

The music dataset was downloaded via webscraping from the online database MusicBrainz[8]. The webscraping itself was implemented in a

[7] https://www.mopop.org/
[8] https://musicbrainz.org/doc/MusicBrainz_API

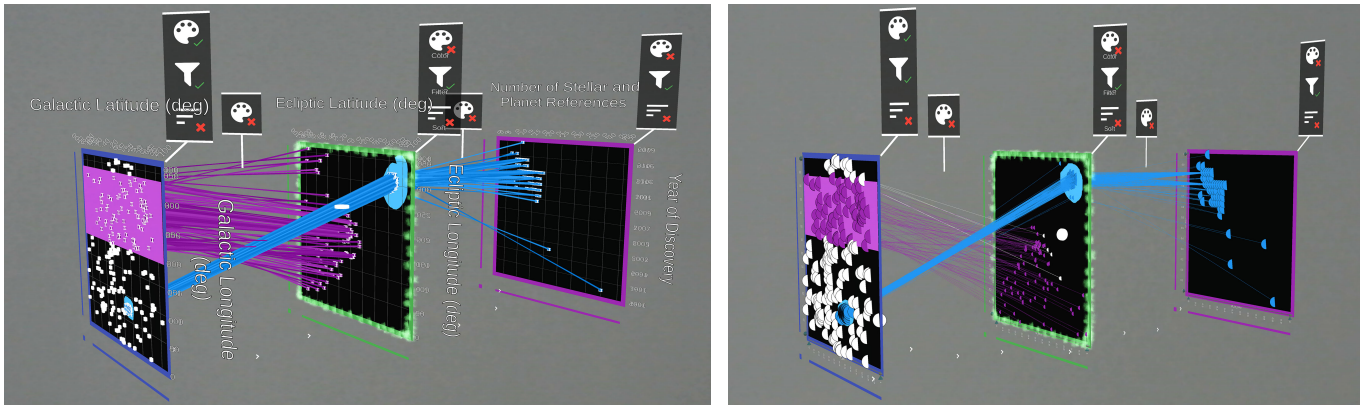

Fig. 14. (left) 3D parallel coordinates visualization in the original implementation of STREAM, (right) the same visualization recreated with the re-implementation in RagRug. In both visualizations, data is progressively filtered from the leftmost to the rightmost scatterplot, while the color is determined by the first scatterplot.

few hours using Node-RED (Figure 12). MusicBrainz data could be retrieved on the fly, but we cache the relevant data in Postgres for faster cross-referencing with the exhibits.

Implementation The curator is equipped with AR glasses (HoloLens) presenting several interactive visualizations. *Preview*: Upon picking up an exhibit, its attributes are shown as a textual overlay. *Occupancy grid*: When an exhibit is attached to a wall, it physically occupies a varying amount of space – CD covers take 18×18 cm; vinyl record sleeves take 30×30 cm, and posters occupy 30×50 cm. Free grid positions on the exhibition walls are visualized in grey and turn green when occupied. *Timeline*: Below each exhibition wall, a virtual timeline is presented to help with keeping the chronological order (Figure 11). Exhibits in non-chronological order will create an intersection of leader lines, instantly conveying poor placement to the curator. *Year distribution*: A histogram of the years of all exhibits on the wall is shown perpendicular to each exhibition wall (Figure 11).

MoPop uses image recognition[9] to detect and track 100 exhibits (the maximum number allowed by the recognition library). When the user picks up an exhibit, and it is recognized by the camera on the AR device, a message is published to inform other components about the curator's current interest.

Each grid position on the exhibition walls is connected to a collider object in Unity. When the user attaches an exhibit, the collider triggers multiple updates: First, feedback (guidance) on the placement is determined and published. Second, the year histogram for each wall is updated, and the new histogram is re-published. Third, a leader-line is created to connect each exhibit to its year on the timeline. Fourth, subscribers to the mentioned topics receive the updates and refresh their content: Year histograms and feedback visualizations get updated.

5.3 STREAM: collaborative immersive analytics

Scenario STREAM is a recently published, sophisticated application for immersive analytics in AR, which is especially interesting to us because it supports collaborative work [32]. In STREAM, users can arrange multiple 2D scatterplots in 3D space and link them together to form 3D parallel coordinate trees. Brushing, filtering, and other manipulations of the visualization can be conducted either directly in 3D or via touch in 2D, on a spatially-aware tablet given to each user. STREAM has complex visualizations and a sophisticated interface for manipulating them. However, it has no referents and therefore addresses a "pure" immersive analytics scenario. Yet, STREAM is a multi-device and multi-user application with demanding requirements for coordinating distributed software and hardware components. Therefore, we expected that a re-implementation of STREAM on top of RagRug would reveal if RagRug is able to address the needs of immersive analytics in addition to the needs of situated analytics.

[9] `https://vuforia.com`

Physical-virtual model Since STREAM is an immersive analytics application, it does not include any referents and has a trivial spatial model, which only helps to establish a common coordinate system between multiple concurrent users. STREAM visualizes pre-configured tabular data, which can simply be retrieved via HTTP. However, a relational database such as Postgres could be used to support more sophisticated query and filtering operations on larger data collections.

Implementation The original STREAM implements multi-user synchronization with reactive event handling between users. This approach requires writing application code that explicitly sends messages to notify other users about changes to the visualization. We decided that it would be more efficient to add automatic replication to IATK instead. With a replicated IATK approach, a user can change the local visualization, and the remote replicas of the visualization will be automatically synchronized.

Since IATK is a plug-in of Unity, other projects using IATK (e.g., FIESTA [43]) have relied on Unity networking libraries to synchronize the Unity game objects underlying IATK. Since there are several widely used libraries for game object sharing, such as Photon[10] or Mirror[11], using them seems straightforward. Yet, these libraries have several disadvantages for our scenario: Treating IATK visualizations as generic game objects in synchronization sacrifices our understanding of what exactly has changed inside the visualization. If IATK caches data on the GPU, pure synchronization of game objects stored in CPU memory may not even result in a working solution. Even if game object-level synchronization did work correctly, it would likely increase latency and require coarse-grained locking to avoid race conditions between users. Moreover, we would like to avoid adding another heavyweight component (a third-party Unity networking library) to our toolkit.

We found that replication of IATK visualizations is a good alternative, especially since it can be built atop the existing pub-sub. We establish synchronization by making the replica a subscriber of a master visualization. When a change is made to IATK, an observer object [22] serializes the change into the JSON format (including its data sources).

The serialized updates are published by the master visualization. Replicas subscribe to these messages and apply the content to their replicated visualization via the reactive IATK extension described in Section 4.4. Replicas can either be local (residing in the same Unity process space) or remote (residing on another host in the network, e.g., a second AR device). Remote replicas rely on MQTT for receiving updates, thereby re-using the existing network infrastructure. To ensure minimal latency, streams from external sources (e.g., IoT sensors) are received by all replicas directly and do not need to be relayed through the master visualization.

We have used this IATK replication to create a new version of

[10] `https://www.photonengine.com/`
[11] `https://mirror-networking.com`

Fig. 15. (left) The ControllAR application overlays a visualization of the envelope curve on the corresponding inputs on the keyboard. (right) The responsibilities for this application are distributed across the keyboard (referent and physical input device), the MIDI client (responsible for sound synthesis and for converting MIDI events to MQTT events), the hub (here, only acting as a relay) and the AR client (compute envelope visualization and display it registered in the space above the keyboard).

STREAM, replacing the custom visualizations of STREAM with IATK and the custom synchronization of STREAM with the IATK replication. When creating a visualization (e.g., by pressing a button on a tablet), a matching IATK specification is created in Node-RED and sent to all clients. Upon receiving, each client builds the IATK visualization. From here on, all updates are automatically distributed using the IATK replication, with little additional logic necessary. We use the IATK replication in a symmetric configuration, which treats each visualization as master concerning changes made by the client's user, and, as replica for changes made by other users.

STREAM uses VIVE Lighthouse sensors for tracking the tablet's spatial position. Although these sensors send their data using proprietary messages, we can handle them similar to other IoT devices in RagRug. We convert them on the fly into MQTT messages to deliver the tracking data in the expected format. Thus, clients can subscribe to the tracking data stream (e.g., for calculating the tablet's orientation) without requiring any additional case distinctions in Node-RED or in the client code. Proprietary tracking sensors become just another data source in RagRug.

With the strategy described above, the RagRug toolkit supplanted the proprietary client/server architecture of the original STREAM architecture. The behavior of the STREAM server could be entirely reproduced with Node-RED and MQTT in a few hours. A particularly efficient tool was the pub-sub routing, which let us replace the custom routing code of the original STREAM server with appropriately chosen topics. Another useful feature was that the current application state of STREAM could be saved to a database with a few dataflow nodes. Additional features (e.g., data filter logic) were added as dataflow nodes with custom Javascript that reacts to specific events.

Special visualization and interaction features used in STREAM (e.g., colored borders around scatter plots, hitboxes for application logic) were added as additional metadata to the IATK visualization specification, so they could be piggybacked to the replica update mechanism. As a result, we achieved a look and feel equivalent to the original STREAM implementation (Figure 14) with a fraction of the original code and complexity (reducing the lines of C# code responsible for the visualization alone from 2956 to 1878).

5.4 ControllAR: musical instrument control

Scenario *ControllAR* [36] is an AR visualization system for MIDI devices, which supports musicians in the use of a keyboard instrument. In the original implementation of ControllAR, 2D visualizations generated by a static display are overlaid via a half-way mirror onto the keyboard, so that the visualizations are registered with physical buttons, sliders, knobs and other input elements. The visualizations are reactive to the user's input on the keyboard, which is connected to the host computer via the keyboard's MIDI interface. We chose this scenario to demonstrate fine-grained, precise embedding of interactive visualizations in the coordinate frame of a referent.

Physical-virtual model Our physical-virtual model consists of the keyboard model, the visualization templates (including data sources to connect to), the mapping between keyboard parts and visualization templates, and, of course, the application logic in Javascript. As before, this data is stored in CouchDB on the hub and downloaded by the AR client on demand.

Implementation Our re-implementation of ControllAR uses RagRug to gather, process, and visualize data generated by a MIDI keyboard (Komplete Kontrol M32[12]). The keyboard is connected via USB to a MIDI client. For convenience, we used a desktop computer as MIDI client, but an inexpensive single-board computer could be used instead to create a self-contained smart keyboard. The MIDI client connects to the hub, and a HoloLens is used as the AR client in the same manner as in our other scenarios (Figure 15).

In AR, we visualize the envelope curve corresponding to the selected equalizer settings. The visualization responds in real time to manipulations of the amplitude envelope and frequency envelope of the current sound. The visualizations are placed relative to a hierarchical geometric model of the keyboard and its relevant input elements. Since the keyboard is a movable referent, we register it from a small image marker affixed to the keyboard.

The sound parameters are manipulated using 2×4 dedicated knobs on the keyboard. When changed, each knob sends MIDI events to the MIDI client, which re-publishes them as MQTT messages. The MIDI client also generates the audio output using a synthesizer application written in Javascript[13].

The AR client subscribes to the messages published by the MIDI client and converts them into an IATK visualization of the amplitude and frequency envelope curve. The envelope curves are shown adjacent to the keyboard in Figure 15, serving as an expansion of the keyboard's tiny built-in display.

6 Performance Measurements

We complement the qualitative case studies with a brief review of performance characteristics. The display of our main output device, the HoloLens 2, uses dedicated hardware to ensure a refresh rate of 30 Hz for rendering of the Unity scene graph containing the IATK visualizations from a head-tracked viewpoint. However, changes to the visualization other than viewport changes require processing of dataflow events. This processing is subject to the run-time interpretation of Javascript code and event distribution in the network. Delayed events can increase the latency of interactive experiences.

Therefore, we were interested in the performance characteristics of event propagation in RagRug. We connected a HoloLens 2 and a notebook computer (Intel Core i7-7700HQ, 16B Ram) acting as Hub via an of-the-shelf Wifi router (Netgear R6400, 5 Ghz band, 802.11a/n/ac). The wireless network was shared with other computers and devices, so the available bandwidth is variable. We report average times measured over a duration of several minutes.

We tested two types of event propagation: Local event propagation on the HoloLens client sends the event, either by passing events through

[12] https://www.native-instruments.com/de/products/komplete/keyboards/komplete-kontrol-m32/
[13] https://tonejs.github.io

dataflow nodes or using local pub-sub. Since both types of local propagation are technically mapped to Javascript function invocation, their performance differences are negligible. Round-trip event propagation originates at the HoloLens client, is relayed via Mosquitto on the hub, and then received and delivered at the HoloLens again. While round-trip events take an average of 150 ms (minimum: 100 ms, maximum 300 ms), local events are delivered in under 1.5 ms. At 30 Hz framerate, delay of local events is generally unnoticeable.

Event throughput on the HoloLens 2 is unaffected even if events are propagated locally at rates that are multiple times higher than the framerate. The only noticeable costly operation in the Unity framework is write access to game objects, which is internally serialized through single-threaded co-routines. The runtime penalty of the dataflow is negligible as long as the number of game object accesses is minimized. Since our applications target a particular game object representing IATK, RagRug aggregates all events intended for IATK and delivers them exactly at framerate to avoid any slowdowns. This strategy ensures that IATK applications are unaffected by the single-threaded event delivery, as long as no other Unity game objects are involved. On faster hardware, such as an iPad or a desktop computer, we found this optimization to be unnecessary.

7 Discussion

The significance of a software toolkit is directly related to its acceptance. In this section, we describe some of the experiences the authors of this paper have made as developers using RagRug, and, statements made by other researchers in discussions about RagRug.

One of our main intents was supporting rapid prototyping. We have characterized "situated authoring" as an activity that rapidly switches between code writing, content creation, and testing in a physical (or semi-physical) environment. Concentrating on the non-coding activities means having less time for the coding activities. For this reason, we adopted dataflow-oriented programming, which, together with the use of visual programming, is favored in IoT development. Unsurprisingly, we found that this programming style offers similar benefits to AR development. Not only can code be easily modified on the fly with RagRug, it can be deployed across the distributed system just as easily. Activities traditionally associated with development on embedded systems, such as compiling, packaging, and deploying code, then restarting the device, are entirely avoided. Moreover, the RagRug infrastructure subsumes a lot of boilerplate code that would otherwise be required to start up a system consisting of several distributed components across multiple hardware platforms and operating systems.

These savings were particularly obvious when comparing the original version of STREAM to the RagRug version of the same application. Large portions of the original code, including the network communication and the server infrastructure, could be replaced with toolkit functions. As a byproduct of relying on the toolkit, a variety of features (e.g., multi-user replication or logging of events) is either free or requires little additional effort. Besides, breaking the monolithic interaction code of the original STREAM implementation into separate dataflows also helped to modularize the code, making it easier to maintain in the future. In the dataflow of RagRug, communication endpoints are explicitly visible, which helps understanding the system's behavior. It is also straightforward to insert debugging facilities as extra nodes at critical places in the dataflow.

The emphasis on low-code programming comes, to some extent, at the expense of code scalability. Complex coding is better left to traditional development environments, which excel at supporting developers in navigating large amounts of code, debugging using breakpoints, or interfacing with version control systems, such as Git. The strive for balance between ease of use and expressiveness regarding development tools reminds of the long-standing dispute in the software engineering community between advocates of micro-services vs monolithic applications. The issue is known to the Node-RED community as well. Indeed, alternative development environments specifically targeting large Node-RED projects are under active investigation[14]. Since RagRug seamlessly integrates into the development suite of Node-RED, it benefits from such efforts addressing scalability of developing with Node-RED. Moreover, it is already entirely feasible with the current state of RagRug to write the core application logic in a monolithic style, while using RagRug only for the user interface aspects.

Another aspect that may require some acclimatization time for seasoned developers is proper timing of the dataflow. Lengthy computations can stall the dataflow and lead to congestion of buffers holding data to be processed. Multi-threading can be used to accelerate expensive computations in the dataflow, since the scheduling of dataflow nodes is asynchronous. However, using multi-threading may not be entirely straightforward, since it is subject to two conditions: First, enough computational resources must be available; second, all the required data must be stored in local memory of the multi-processor. On mobile AR devices, computational performance may be insufficient, and a round trip of the data to the hub may have excessive latency. These restrictions are familiar to developers of distributed IoT systems, but VR/AR developers may be less used to them.

[14] `https://nodered.org/docs/user-guide/projects/`

8 Conclusions and future work

Our examples show that the complex emerging domain of situated analytics applications can be supported by a toolkit which integrates referents into the visualization pipeline. Such an integration must go beyond merely transposing visualization and interaction from 2D to 3D. Indeed, the focus of RagRug is not on 3D visualization or interaction itself. Instead, it lets the user set up a distributed dataflow system upon which a situated visualization pipeline can be built.

While we have successfully used RagRug to build a variety of application prototypes, the work on the toolkit is far from complete. The current version of RagRug provides only the low-level features for making visualizations reactive to the environment. In other words, RagRug provides ample mechanisms, but hardly any policies for reactive visualizations. There is a substantial body of work on handling occlusion, clutter, contrast, temporal coherence, and other artefacts of mixed reality displays. Since previous work in this area generally lacks re-usable implementations, there is limited practical experience with AR visualization algorithms. Yet, many issues of situated visualization (and spatial visualization in general) require more and broader experimentation [76]. With a toolkit such as RagRug, the effort in conducting such experiments becomes substantially more manageable.

RagRug and the examples are available on Github:
`https://github.com/philfleck/ragrug`

Acknowledgments

This research was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2120/1 – 390831618, as well as by the DFG Project-ID 251654672 – TRR 161.

References

[1] Vega-Lite – A Grammar of Interactive Graphics. `https://vega.github.io/vega-lite/`. Last Accessed: 2021-03-28.

[2] C. Andrews and C. North. Analyst's workspace: An embodied sensemaking environment for large, high-resolution displays. In *IEEE Conference on Visual Analytics Science and Technology (VAST)*, pp. 123–131, 2012. doi: 10.1109/VAST.2012.6400559

[3] M. Back, D. Kimber, E. Rieffel, A. Dunnigan, B. Liew, S. Gattepally, J. Foote, J. Shingu, and J. Vaughan. The virtual chocolate factory: Building a real world mixed-reality system for industrial collaboration and control. In *IEEE International Conference on Multimedia and Expo (ICME)*, pp. 1160–1165, 2010. doi: 10.1109/ICME.2010.5582532

[4] S. K. Badam, F. Amini, N. Elmqvist, and P. Irani. Supporting visual exploration for multiple users in large display environments. In *IEEE Conference on Visual Analytics Science and Technology (VAST)*, pp. 1–10, mar 2017. doi: 10.1109/VAST.2016.7883506

[5] S. K. Badam and N. Elmqvist. Visfer: Camera-based visual data transfer for cross-device visualization. *Information Visualization*, 18(1):68–93, 2019. doi: 10.1177/1473871617725907

[6] R. Ball and C. North. Realizing embodied interaction for visual analytics through large displays. *Computers and Graphics (Pergamon)*, 31(3):380–400, 2007. doi: 10.1016/j.cag.2007.01.029

[7] M. Beaudouin-Lafon. Instrumental interaction: An interaction model for designing post-wimp user interfaces. In *ACM CHI*, pp. 446–453, 2000. doi: 10.1145/332040.332473

[8] P. Belimpasakis and R. Walsh. A combined mixed reality and networked home approach to improving user interaction with consumer electronics. *IEEE Transactions on Consumer Electronics*, 57(1):139–144, 2011. doi: 10.1109/TCE.2011.5735494

[9] J. Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983.

[10] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE TVCG*, 15(6):1121–1128, 2009. doi: 10.1109/TVCG.2009.174

[11] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE TVCG*, 17:2301–2309, 2011. doi: 10.1109/TVCG.2011.185

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley, 1996.

[13] P. W. S. Butcher, N. W. John, and P. D. Ritsos. VRIA: A web-based framework for creating immersive analytics experiences. *IEEE TVCG*, pp. 1–1, 2020. doi: 10.1109/TVCG.2020.2965109

[14] Z. Chen, Y. Su, Y. Wang, Q. Wang, H. Qu, and Y. Wu. MARVisT: Authoring glyph-based visualization in mobile augmented reality. *IEEE TVCG*. doi: 10.1109/TVCG.2019.2892415

[15] Z. Chen, W. Tong, Q. Wang, B. Bach, and H. Qu. Augmenting static visualizations with PapARVis designer. *ACM CHI*, 2020. doi: 10.1145/3313831.3376436

[16] M. Cordeil, A. Cunningham, B. Bach, C. Hurter, B. H. Thomas, K. Marriott, and T. Dwyer. IATK: An immersive analytics toolkit. In *IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 200–209, 2019. doi: 10.1109/VR.2019.8797978

[17] N. Elmqvist, A. V. Moere, H.-C. Jetter, D. Cernea, H. Reiterer, and T. Jankun-Kelly. Fluid interaction for information visualization. *Information Visualization*, 10(4):327–340, 2011. doi: 10.1177/1473871611413180

[18] N. ElSayed, B. Thomas, K. Marriott, J. Piantadosi, and R. Smith. Situated Analytics. *Big Data Visual Analytics*, 2015. doi: 10.1109/BDVA.2015.7314302

[19] B. Ens, F. Anderson, T. Grossman, M. Annett, P. Irani, and G. Fitzmaurice. Ivy: Exploring spatially situated visual programming for authoring and understanding intelligent environments. *Graphics Interface*, pp. 156–162, 2017. doi: 10.20380/gi2017.20

[20] O. Erat, W. A. Isop, D. Kalkofen, and D. Schmalstieg. Drone-augmented human vision: Exocentric control for drones exploring hidden areas. *IEEE TVCG*, 24(4):1437–1446, 2018. doi: 10.1109/TVCG.2018.2794058

[21] J.-D. Fekete. The InfoVis Toolkit. In *IEEE Symposium on Information Visualization*, pp. 167–174, 2004. doi: 10.1109/INFVIS.2004.64

[22] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[23] M. Gandy and B. MacIntyre. Designer's augmented reality toolkit, ten years later: Implications for new media authoring tools. In *ACM UIST*, pp. 627–636, 2014. doi: 10.1145/2642918.2647369

[24] J. A. Garcia-Macias, J. Alvarez-Lozano, P. Estrada, and E. Aviles Lopez. Browsing the internet of things with sentient visors. *IEEE Computer*, 44(5):46–52, 2011. doi: 10.1109/MC.2011.128

[25] J. Grubert, M. Pahud, M. Kranz, and D. Schmalstieg. GlassHands: Interaction around unmodified mobile devices using sunglasses. In *ACM Interactive Surfaces and Spaces (ISS)*, 2016. doi: 0.1145/2992154.2992162

[26] A. S. Gunnarsson, M. Rauhala, A. Henrysson, and A. Ynnerman. Visualization of sensor data using mobile phone augmented reality. In *IEEE ISMAR*, pp. 233–234, 2007. doi: 10.1109/ISMAR.2006.297820

[27] C. Harrison, H. Benko, and A. D. Wilson. OmniTouch: Wearable multitouch interaction everywhere. In J. S. Pierce, M. Agrawala, and S. R. Klemmer, eds., *ACM UIST*, pp. 441–450, 2011. doi: 10.1145/2047196.2047255

[28] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *ACM CHI*, p. 421–430, 2005. doi: 10.1145/1054972.1055031

[29] B. Herbert, B. Ens, A. Weerasinghe, M. Billinghurst, and G. Wigley. Design considerations for combining augmented reality with intelligent tutors. *Computers and Graphics (Pergamon)*, 77:166–182, 2018. doi: 10.1016/j.cag.2018.09.017

[30] D. Herr, J. Reinhardt, R. Krueger, G. Reina, and T. Ertl. Immersive visual analytics for modular factory layout planning. In *IEEE Immersive Analytics Workshop*, 2017. doi: 0.1016/j.procir.2018.03.200

[31] G. Hillar. *MQTT Essentials - A Lightweight IoT Protocol*. Packt Publishing, 2017.

[32] S. Hubenschmid, J. Zagermann, S. Butscher, and H. Reiterer. STREAM: Exploring the combination of spatially-aware tablets with augmented reality head-mounted displays for immersive analytics. In *ACM CHI*, 2021. doi: 10.1145/3411764.3445298

[33] K. Huo, Y. Cao, S. H. Yoon, Z. Xu, G. Chen, and K. Ramani. Scenariot: Spatially mapping smart things within augmented reality scenes. In *ACM CHI*, 2018. doi: 10.1145/3173574.3173793

[34] Y. Jansen and P. Dragicevic. An interaction model for visualizations beyond the desktop. *IEEE TVCG*, 19(12):2396–2405, 2013. doi: 10.1109/TVCG.2013.134

[35] D. Jo and G. J. Kim. ARIoT: Scalable augmented reality framework for interacting with Internet of Things appliances everywhere. *IEEE Transactions on Consumer Electronics (TCE)*, 62(3):334–340, 2016. doi: 10.1109/TCE.2016.7613201

[36] A. Jones and F. Berthaut. Controllar: Appropriation of visual feedback on control surfaces. In *ACM Conference on Interactive Surfaces and Spaces (ISS)*, p. 465–468, 2016. doi: 10.1145/2992154.2998580

[37] S. Kasahara, R. Niiyama, V. Heun, and H. Ishii. ExTouch: Spatially-aware embodied manipulation of actuated objects mediated by augmented reality. In *ACM TEI*, pp. 223–226, 2013. doi: 10.1145/2460625.2460661

[38] G. R. King, W. Piekarski, and B. H. Thomas. ARVino - Outdoor augmented reality visualisation of viticulture GIS data. In *IEEE ISMAR*, pp. 52–55, 2005. doi: 10.1109/ISMAR.2005.14

[39] J. Lacoche, T. Duval, B. Arnaldi, E. Maisel, and J. Royan. A survey of plasticity in 3D user interfaces. In *IEEE Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 19–26, 2014. doi: 10.1109/SEARIS.2014.7152797

[40] J. I. Larregui, M. Luján Ganuza, E. A. Bjerg, S. Castro, N. As, F. Gazcón, G. Gazcón, J. M. Trippel Nagel, M. L. Ganuza, and S. M. Castro. Immersive analytics for geology: Field sketch-like visualization to assist geological structure analysis during fieldwork. Technical report, 2018.

[41] F. Ledermann and D. Schmalstieg. APRIL – A high level framework for creating augmented reality presentations. In *IEEE VR*, pp. 187–194, 2005. doi: 10.1109/VR.2005.1492773

[42] D. Ledo, S. Greenberg, N. Marquardt, and S. Boring. Proxemic-aware controls: Designing remote controls for ubiquitous computing ecologies. In *Proc. MobileHCI*, pp. 187–198, 2015. doi: 10.1145/2785830.2785871

[43] B. Lee, X. Hu, M. Cordeil, A. Prouzeau, B. Jenny, and T. Dwyer. Shared surfaces and spaces: Collaborative data visualisation in a co-located immersive environment. *IEEE TVCG*, 27(2):1171–1181, 2021. doi: 10.1109/TVCG.2020.3030450

[44] B. Lee, P. Isenberg, N. H. Riche, and S. Carpendale. Beyond mouse and keyboard: Expanding design considerations for information visualization interactions. *IEEE TVCG*, 18(12):2689–2698, 2012. doi: 10.1109/TVCG.2012.204

[45] Z. Li, M. Annett, K. Hinckley, K. Singh, and D. Wigdor. Holodoc: Enabling mixed reality workspaces that harness physical and digital content. In *ACM CHI*, pp. 1–14, 2019. doi: 10.1145/3290605.3300917

[46] C. Liu, S. Huot, J. Diehl, W. E. MacKay, and M. Beaudouin-Lafon. Evaluating the benefits of real-time feedback in mobile Augmented Reality with hand-held devices. In *ACM CHI*, pp. 2973–2976, 2012. doi: 10.1145/2207676.2208706

[47] B. MacIntyre, A. Hill, H. Rouzati, M. Gandy, and B. Davidson. The argon ar web browser and standards-based ar application environment. In *IEEE ISMAR*, pp. 65–74, 2011. doi: 10.1109/ISMAR.2011.6092371

[48] A. MacWilliams, C. Sandor, M. Wagner, M. Bauer, G. Klinker, and B. Bruegge. Herding sheep: Live system development for distributed augmented reality. In *IEEE ISMAR*, 2003. doi: 10.1109/ISMAR.2003.1240695

[49] L. Merino, B. Sotomayor-Gómez, X. Yu, R. Salgado, A. Bergel, M. Sedlmair, and D. Weiskopf. Toward agile situated visualization: An exploratory user study. In *CHI Extended Abstracts*, 2020. doi: 10.1145/3334480.3383017

[50] P. Milgram and F. Kishino. A taxonomy of mixed reality visual displays. *IEICE Transactions on Information Systems*, E77-D(12):1321–1329, 1994. doi: 10.1.1.102.4646

[51] M. Nebeling, M. Speicher, X. Wang, S. Rajaram, B. D. Hall, Z. Xie, A. R. E. Raistrick, M. Aebersold, E. G. Happ, J. Wang, Y. Sun, L. Zhang, L. E. Ramsier, and R. Kulkarni. MRAT: The mixed reality analytics toolkit.

In *ACM CHI*, p. 1–12, 2020. doi: 10.1145/3313831.3376330
[52] N. Norouzi, G. Bruder, B. Belna, S. Mutter, D. Turgut, and G. Welch. A systematic review of the convergence of augmented reality, intelligent virtual agents, and the internet of things. In *Artificial Intelligence in IoT*. 2019. doi: 10.1007/978-3-030-04110-6_1
[53] D. R. Olsen. Evaluating user interface systems research. In *ACM UIST*, pp. 251–258, 2007. doi: 10.1145/1294211.1294256
[54] B. Pokric, S. Krco, and M. Pokric. Augmented reality based smart city services using secure IoT infrastructure. In *IEEE Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pp. 803–808, 2014. doi: 10.1109/WAINA.2014.127
[55] A. Prouzeau, Y. Wang, B. Ens, W. Willett, and T. Dwyer. Corsican Twin: Authoring in situ augmented reality visual-isations in virtual reality. In *International Conference on Advanced Visual Interfaces (AVI)*, 2020. doi: 10.1145/3399715.3399743
[56] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: An open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.
[57] P. Reipschlager, T. Flemisch, and R. Dachselt. Personal augmented reality for information visualization on large interactive displays. *IEEE TVCG*, 27(2):1182–1192, 2021. doi: 10.1109/TVCG.2020.3030460
[58] G. Reitmayr and D. Schmalstieg. An open software architecture for virtual reality interaction. In *ACM VRST*, p. 47–54, 2001. doi: 0.1145/505008.505018
[59] D. Salber, A. K. Dey, and G. D. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *ACM CHI*, p. 434–441, 1999. doi: 10.1145/302979.303126
[60] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE TVCG*, 22(1):659–668, jan 2016.
[61] D. Schmidt, D. Molyneaux, and X. Cao. PICOntrol: Using a handheld projector for direct control of physical devices through visible light. p. 379, 2012. doi: 10.1145/2380116.2380166
[62] R. Sicat, J. Li, J. Choi, M. Cordeil, W. K. Jeong, B. Bach, and H. Pfister. DXR: A toolkit for building immersive data visualizations. *IEEE TVCG*, 25(1):715–725, 2019. doi: 10.1109/TVCG.2018.2865152
[63] J. C. Spohrer. Information in places. *IBM Systems Journal*, 38(4):602–628, Dec. 1999. doi: 10.1147/sj.384.0602
[64] H. Subramonyam, S. M. Drucker, and E. Adar. Affinity lens data-assisted affinity diagramming with augmented reality. In *ACM CHI*, pp. 1–13, 2019. doi: 10.1145/3290605.3300628
[65] R. Suzuki, K. Masai, and M. Sugimoto. ReallifeEngine: A mixed reality-based visual programming system for smarthomes. *ICAT-EGVE*, 2019. doi: 10.2312/egve.20191287
[66] R. M. Taylor, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. VRPN: A device-independent, network-transparent vr peripheral system. In *ACM VRST*, p. 55–61, 2001. doi: 10.1145/505008.505019
[67] B. H. Thomas, G. F. Welch, P. Dragicevic, N. Elmqvist, P. Irani, Y. Jansen, D. Schmalstieg, A. Tabard, N. A. M. ElSayed, R. T. Smith, and W. Willett. *Situated Analytics*, pp. 185–220. Springer International Publishing, 2018. doi: 10.1007/978-3-030-01388-2_7
[68] E. Veas, R. Grasset, I. Ferencik, T. Gruenewald, and D. Schmalstieg. Mobile Augmented Reality for Environmental Monitoring. *Personal and Ubiquitous Computing*, 2012. doi: 10.1007/s00779-012-0597-z
[69] J. A. Walsh and B. H. Thomas. Visualising environmental corrosion in outdoor augmented reality. In *Australasian User Interface Conference*, pp. 39–46, 2011.
[70] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Lecture Notes in Computer Science*, vol. 2257, pp. 155–172. Springer Verlag, 2002.
[71] S. White and S. Feiner. Sitelens: Situated visualization techniques for urban site visits. In *ACM CHI*, 2009. doi: 10.1145/1518701.1518871
[72] W. Willett, Y. Jansen, and P. Dragicevic. Embedded data representations. *IEEE TVCG*, 23(1):461–470, 2017. doi: 10.1109/TVCG.2016.2598608
[73] R. Xiao, C. Harrison, and S. E. Hudson. WorldKit: Rapid and easy creation of ad-hoc interactive applications on everyday surfaces. In *ACM CHI*, p. 879, 2013. doi: 10.1145/2470654.2466113
[74] R. Xiao, J. Schwarz, N. Throm, A. D. Wilson, and H. Benko. MRTouch: Adding touch input to head-mounted mixed reality. *IEEE TVCG*, 24(4):1653–1660, 2018. doi: 10.1109/TVCG.2018.2794222
[75] J. S. Yi, Y. A. Kang, J. T. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE TVCG*, 13(6):1224–1231, nov 2007. doi: 10.1109/TVCG.2007.70515
[76] B. Yost and C. North. The perceptual scalability of visualization. In *IEEE TVCG*, vol. 12, pp. 837–844, sep 2006. doi: 10.1109/TVCG.2006.184
[77] L. Zhang and S. Oney. FlowMatic: An immersive authoring tool for creating interactive scenes in virtual reality. In *ACM UIST*, p. 342–353, 2020. doi: 10.1145/3379337.3415824
[78] M. Zhao, Y. Su, J. Zhao, S. Chen, and H. Qu. Mobile situated analytics of ego-centric network data. In *ACM SIGGRAPH Asia Symposium on Visualization*, 2017. doi: 10.1145/3139295.3139309
[79] S. Zollmann, C. Hoppe, T. Langlotz, and G. Reitmayr. FlyAR: augmented reality supported micro aerial vehicle navigation. *IEEE TVCG*, 20(4):560–8, 2014. doi: 10.1109/TVCG.2014.24
[80] S. Zollmann, D. Kalkofen, E. Mendez, and G. Reitmayr. Image-based ghostings for single layer occlusions in augmented reality. In *IEEE ISMAR*, pp. 19–26, 2010. doi: 10.1109/ISMAR.2010.5643546

Philipp Fleck is a PhD candidate at Graz University of Technology. His work focuses on Augmented Reality, Visualization and distributed real-time systems. He received his Master's degree in 2015 from Graz University of Technology.

Aimée Sousa Calepso is a PhD student at the University of Stuttgart at the Visualization Research Center. She received her Master's degree from the Federal University of Rio Grande do Sul in 2020. In her research she focuses on immersive visualizations, usability assessments and interactions in VR.

Sebastian Hubenschmid is a PhD student at the HCI Group at the University of Konstanz, Department of Computer and Information Science. He received his Master's degree in 2019 in Konstanz. In his research, he focuses on the use of transitional and hybrid user interfaces for immersive analytics.

Michael Sedlmair is a professor at the University of Stuttgart, where he works at the intersection of human-computer interaction, visualization, and data analysis. His specific research interests focus on information visualization, interactive machine learning, virtual and augmented reality, and evaluation methodologies.

Dieter Schmalstieg (Fellow, IEEE) is a professor at Graz University of Technology, Austria. His research interests span augmented reality, virtual reality, and visualization. He has been associate editor in chief of TVCG and is a recipient of the IEEE Virtual Reality technical achievement award.