Alpaca: AR Graphics Extensions for Web Applications

Tanner Hobson* Jeremiah Duncan* Mohammad Raji* Aidong Lu† Jian Huang*

ABSTRACT

In this work, we propose a framework to simplify the creation of Augmented Reality (AR) extensions for web applications, without modifying the original web applications. We implemented the framework in an open source package called Alpaca. AR extensions developed using Alpaca appear as a web-browser extension, and automatically bridge the Document Object Model (DOM) of the web with the SceneGraph model of AR. To transform the web application into a multi-device, mixed-space web application, we designed a restrictive and minimized interface for cross-device event handling. We demonstrate our approach to develop mixed-space applications using three examples. These applications are, respectively, for exploring Google Books, exploring biodiversity distribution hosted by the National Park Service of the United States, and exploring YouTube's recommendation engine. The first two cases show how a 3rd-party developer can create AR extensions without making any modifications to the original web applications. The last case serves as an example of how to create AR extensions when a developer creates a web application from scratch. Alpaca works on the iPhone X, the Google Pixel, and the Microsoft HoloLens.

Index Terms: Computing methodologies—Computer graphics—Graphics systems and interfaces—Mixed / augmented reality; Computer systems organization—Architectures—Distributed architectures—Client-server architectures;

1 INTRODUCTION

Since inception, the web has led to a rich and diverse ecosystem for developing applications that impact our daily lives. This ecosystem lives in our 2D screens. With the advent of AR, researchers have attempted marrying AR with web in custom-built applications to leverage the rich interactions and explorability of AR.

Rewriting web applications with AR in mind can be costly. There are few overlaps between the web development and the AR development worlds. The near ubiquity of web browsers and the availability of a standard executing environment contrasts sharply with the niche and disparate ecosystems that AR applications must conform to. In essence, adding AR to an existing application has until now meant a costly rewrite and rearchitecture.

In this work, we have developed a bridging framework that simplifies extending web applications into AR without a complete rearchitecture. Our framework is called Alpaca and has two main areas of improvement. In terms of **content**, it enables immersive exploration of natively 3D information using an AR device, rather than being constrained to the 2D-screen-space. In terms of **architecture**, Alpaca provides a minimal bridging interface that reduces the complexity and overhead of mixed-space applications.

While some web applications may have AR counterparts, mixed-space web applications developed with Alpaca run on a desktop and an AR device (if present) simultaneously, which extends the spaces in which a user can explore the content in the web application.

We intend Alpaca to be a personalized bridging framework that connects live web and AR contexts on a user's laptop and AR device, respectively. At present, the visual channel of AR is the sole focus of this work. We design that the Alpaca Server, which implements cross-device synchronization, runs on the user's laptop rather than as a publicly accessible service. By running the server locally, we can minimize the cross-device latencies to improve the user experience of interacting with a mixed-space application.

For AR developers, the Alpaca framework provides an abstracted, easy-to-use interface for developers who are not experts of DOM (Document Object Model) to effectively instrument and manage changes to DOM elements. The resulting multi-device capability aids AR developers in significantly lowering their barrier of entry into the widely varied ecosystem of web applications.

For web developers, the Alpaca framework provides a simplified interface and runtime, where code using the web graphics API (e.g. THREE.js) can be automatically bridged to run on an AR device and remain synchronized with the web application. Without needing to program on the AR device, user interactions with the scene graph can auto-update the corresponding web DOM elements in real-time.

For end users, Alpaca appears as a browser extension. When invoked, the Alpaca Server runs as a daemon process and transparently manages interactions between the DOM in the browser and the scene graph in the AR runtime. Users get a synchronized multi-device experience and can interact with Alpaca-enhanced applications both on their laptops and their AR devices, such as iPhone X.

Our design of Alpaca have been driven by three types of application needs: **scene-heavy** applications have large complex scene graph models; **asset-heavy** applications require Alpaca Server to manage many assets concurrently at runtime; **update-heavy** applications incur a sustained high volume of synchronizations.

Our design of Alpaca has been inspired by works such as MapReduce [14] and Tapestry [41, 42] that provide a minimal decoupled interface between the framework's components. Through this work, we have found that it is possible to use a common infrastructure to transparently abstract away all the cross-device communication and synchronization functionalities that are required in event-driven programming. Consequently, the AR device becomes an extension of the web application and becomes application agnostic, enabling reuse.

Our framework is composed of three parts that work in tandem: a browser extension library, a state management server, and an AR device runtime. Each part is in charge of mapping from one type of data model to another. The browser extension maps the DOM to a JSON representation of the 3D scene to be used on the AR device. The server manages this JSON scene structure and synchronizes this scene to the AR device. The AR runtime then takes this scene and manages the rendering and interaction with these objects. AR-initiated events are handled by taking the opposite route.

We demonstrate the efficacy of Alpaca as a reusable infrastructure by building three types of applications: asset-heavy, update-heavy, and scene-heavy applications. These correspond to the Google Books, YouTube, and a GeoMap based application. Our results include performance, memory footprint metrics of Alpaca, and a demo video (in supplemental materials) that show how a user can use the interoperable capabilities to effectively mix the 2D desktop space together with the 3D physical space made available through the AR

*These authors are with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996. E-mail: {thobson2,jdunca51,mahmadza}@vols.utk.edu and huangj@utk.edu

†This author is with the Department of Computer Science, University of North Carolina at Charlotte, NC, 28262. E-mail: aidong.lu@uncc.edu

2642-5254/20/$31.00 ©2020 IEEE
DOI 10.1109/VR46266.2020.00036

devices. In a way, the resulting mixed-space is a new extension of the typical workspace of a user, which deserves much future research by the computer graphics and the VR communities in general.

In the remainder, we cover related work in Section 2, and our system design in Section 3. We describe the process to develop Alpaca applications in Section 4, and then show demonstration applications in Section 5. We conclude in Section 6.

2 Related Work

2.1 Use Scenario

The distributed graphics applications in this work aim to help users use data-intensive web apps from desktop and AR simultaneously.

As web applications have become popular and widespread, many of the most useful web applications often have unique, valuable, and large datasets behind them. While the amount of data and functionality of web applications grow at compounding rates, today's users are also becoming more mobile than before. Oftentimes, a user could be working away from their office or lab settings, where they cannot make use of large displays that have been proven as extremely valuable for tasks that require high density of information [4,5].

We hypothesize that distributed graphics applications can help 2D screen displays of PCs and 3D physical space seen through personal AR devices become an integrated virtual workspace. To that end, while porting web to AR is not a new endeavor, building distributed graphics applications that enable web applications to stride desktop and AR devices simultaneously is a new endeavor.

As follows, let us discuss related works of AR, web-based AR, and cross-device graphics applications in turn.

2.2 Augmented Reality

AR devices are becoming more commonplace and are likely to be an important part of new interfaces. Due to Augmented Reality's native 3D setting in the physical space around the user, 3D data is better conveyed in an AR medium than on a 2D medium [6]. AR can more effectively show relations and trends in graphs [9]. Beyond presenting information, teaching new concepts in AR is also promising [24]. There are many other compelling use cases of AR.

Registration has been an active area of AR research. There were registration systems based on specially designed pictures called markers [11] [47]. However, the required markers introduce difficulties in usability and design of AR scenes [25]. With the advent of more powerful mobile devices and higher resolution cameras and sensors, software-based handheld AR has become popular. Mobile devices now include software to handle this mapping in real time through Google's ARCore [30], Apple's ARKit [22], and Microsoft's HoloLens [13]. These libraries use sensor information from camera images, inertial accelerometers [48], and with the HoloLens, depth map cameras; which together allow the use of AR in complex scenes without any prepared environments or objects. Alpaca's AR on-device runtime uses these software libraries for registration.

AR enables different types of interaction that aren't possible on surface displays. The free control of the AR camera, for instance, allows users to explore inherently 3D data in a natural way compared to a mouse and keyboard. Novel multisensory interaction technologies will let us interact with data in ways that are natural to us and therefore easy to understand [36–38]. Other interactions are still cumbersome in AR, such as text entry or precise data selection and are arguably better with a traditional desktop.

Control of the virtual world in AR often builds on interactions that attempt to mimic the real world. Research shows that some tasks are better performed in AR, such as 3D object manipulation [27]; however, the same is true that some applications are better suited for desktops, such as data filtering and querying tasks [6]. Some tasks that seem better suited towards AR devices, like using the touch screen of a phone for virtual buttons, are implemented in an inflexible or imprecise manner. For example, pressing buttons on a screen moves the camera which alters the view of the scene that was shown.

While AR could increase the perception, cognition, and sensory loads of users [39], studies on using these senses in different applications have shown mixed results [16,21,46,49], however. In addition, we may not need to use all of the senses together [16].

Guidance towards areas of interest in an application is an important aspect of AR applications and one that can build on the existing notion of spatial locality. This area has been explored in AR [17], and many of the insights apply to Alpaca. In desktop applications, guidance is handled through the familiar UI/UX of scrolling and panning through a 2D canvas. Our design philosophy is to help developers offer both forms of guidance in their applications.

2.3 Porting Web to AR

Porting web content or the web ecosystem onto AR devices is not a new topic. Most research so far considers a scenario where an AR device is the sole device of the user. Few have considered building a distributed graphics application where a user has an integrated workspace that spans desktop screen-space and AR physical-space simultaneously. This difference in scope is the primary difference between this work and existing literature.

Showing web content in AR. In this regard, different approaches have been developed for smartphone-based vs head-mounted AR.

On smartphones, due to the prevalence of general-purpose mobile web browsers such as Chrome and Safari, two popular ways to add AR registration capabilities are (i) to make use of AR.js [15], which is a pure-JavaScript marker tracking AR library, or (ii) to build native extensions (e.g. ARCore [30], ARKit [22], or Vuforia [23]) and expose them within the browser as a JavaScript interface. The well known Argon project is a good example [31]. In both cases, rendering still depends on WebGL or THREE.js.

For head-mounted AR, goggle vendors have not prioritized providing AR browsers, for example HoloLens does not have an AR mobile browser capable of showing holograms. In recent years, researchers have created prototype "browsers" of their own [26,31], some with recent AR-oriented optimizations, such as to improve performance by offloading registration tasks into the device's OS [28].

We have designed Alpaca device runtime to provide a universal abstraction, so that these phone-based or head-mounted device details are transparent to application developers.

Authoring AR Content. Authoring AR content is another long-term research priority in the field. For new AR content, some researchers were inspired by web content creation and developed markup languages for developing AR scenes [3,20,43]. Others drew upon the design metaphor of hyper-linked web pages and turned AR scenes into a web of connected areas [26].

For making geospatial data usable in AR, researchers have developed transcoding methods to convert geo-spatial databases into scene graphs [35,52,53]. After content has been transcoded, the AR devices treat the content no differently than if the content was created for AR natively. The geospatial content is primarily rendered as annotations on the 3D physical world [52].

In contrast, we have designed Alpaca to handle general web content, which may or may not include geospatial information. In addition, we design Alpaca's desktop runtime as the dedicated service for transcoding and scene graph creation.

2.4 Collaborative Devices

There are many proven distributed graphics applications that support multi-user as well as single-user multi-device use scenarios. Massive multiplayer online games (e.g. World of Warcraft, Minecraft, Roblox) and collaborative analytics (e.g. Munin [8]) are well known examples. Peer-to-peer architecture has been proven as the most robust and scalable in this field [18,51].

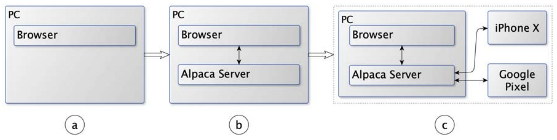

Figure 1: The basic workflow an individual user follows when using Alpaca. In (a), the user interacts with the web application like normal. In (b), the user enables Alpaca which starts up the Alpaca Server and the DOM/Event listener setup process (see Figure 3). In (c), the user starts the Alpaca AR Runtime to see and interact with the AR scene.

Previous works have successfully used both surface and mobile devices [7, 8, 34, 44] together, or AR devices and surface devices together. In particular, to use HoloLens and large-format displays to create multiple-coordinated spaces (MCS) for one user [32] and for collaborative teams of users [33].

In addition, due to precision limits of AR's user interaction, researchers have explored using physical, external devices to improve user control in an AR environment. For examples, MagicBook [10] and Personal Interaction Panel [45] have used physical objects to represent a virtual pen or input device. While these efforts are fruitful, we believe tasks such as text entry and precision selection are still better suited for desktop or other surface screens.

Since users commonly run multiple web applications concurrently in separate browser tabs, in order to lower initiation and application-switching overheads on AR devices, Alpaca focuses on using peer-to-peer state management server to manage states for all of those applications simultaneously. In result, the Alpaca Server can be application-agnostic.

3 Alpaca System Design

We design Alpaca to provide a personalized experience for the user through multiple, cooperative devices on a shared local area network. Let's first illustrate the use case (Figure 1).

To a user, Alpaca's bootstrapping process is as follows: (i) on standard web pages, the user interacts with the page with no change; (ii) on AR-enabled web pages, a notification appears informing them that they can use an AR device for added functionality; (iii) the user opens an app on their phone that starts Alpaca AR runtime and sees AR contents enhanced from the web page.

The user then has two views of the application: on-desktop and in-AR.The desktop views generally remain as they were originally to remain familiar with the user. However, the AR view can be totally novel and supports experimentation into the realm of cross-device use. Some uses include: manipulating the data model of an application for new views of the data (YouTube example); mapping natively 3D data to actual 3D (GeoMaps example); and expanding the user's workspace to incorporate more data (Google Books example).

The extension library exposes listeners to the DOM so that the extension can be alerted when the application state has changed. This enables the development of reactive applications that automatically update based on the changes the user makes in the desktop application without modifying existing application's JavaScript code. The other two main APIs exposed by the extension library are: updating the AR SceneGraph rendered and creating AR listeners on SceneGraph objects. Together, these form the multi-device utilities needed for AR extensions. They will be described more in Section 3.5.

The Alpaca Server runs in the background and awaits HTTP requests from both the browser and the AR runtime. The server maintains an object store of stateful information and makes that store accessible. In addition, it provides a websocket interface that allows the server to notify clients when an object has been modified.

The AR runtime creates a websocket connection to the Alpaca Server and waits for updates to the SceneGraph object in the store. This object is the one created by the extension code. The runtime repeatedly renders this SceneGraph and waits for user interaction to trigger events that will be sent back across the object store to the extension, triggering the callback functions it registered.

To extend a web application to AR, a developer only needs to write the JavaScript extension using THREE.js to place objects in the AR scene. This code is injected into the web application when the user invokes the extension, after which, Alpaca transparently manages cross-device communication and state management.

For performance testing, we focus on the following metrics: (i) AR startup latency, (ii) AR frames per second (FPS), (iii) Alpaca Server footprint (i.e. server memory usage), and (iv) application development complexity (i.e. lines of THREE.js and Alpaca API code). These metrics are discussed in more details in Section 5.

3.1 Overall Architecture

Figure 2 shows a more detailed view of Alpaca's system architecture. We have designed Alpaca using industry-wide standard technologies, including Python's aiohttp library for the Alpaca Server, JavaScript's THREE.js library for the SceneGraph implementation, and built in HTML5 libraries. To the developer, we expose Alpaca like any other HTML5 library for ease of integration.

Alpaca has three main components: Alpaca Server, In-Browser Runtime, and On-Device Runtime.

The Alpaca In-Browser Runtime is triggered through a Chrome extension in the browser. It is application dependent. The Alpaca Server runs persistently and is both application- and user-oblivious. In fact, all demonstration applications in this work use the very same Alpaca Server instance. The server can run on a remote machine for many users and applications or on the user's machine itself for a better guarantee of performance. The On-Device Runtime is application-oblivious. When a user's AR device is on, the device runtime queries the server for the scene specification, and performs the rendering and user-interaction tasks accordingly.

Host Application. The DOM is a key underpinning of all web applications [2]. In essence, the DOM is just an XML model that describes all elements that a web browser renders and presents to each user. While creating an AR extension for a web application, we make absolutely no changes to the host web application.

Alpaca In-Browser Runtime. Alpaca's on-desktop presence is wrapped within a browser extension. This browser extension has different JavaScript scripts that can be injected into the web applications and provide the application-specific changes.

On first load, the Chrome extension contacts the Alpaca Server to register which and how a selected set of DOM objects in the corresponding web application should appear and behave in the on-device AR world. These Alpaca objects are web-session specific. For example, two users using the Google Books extension separately may have the same kinds of objects registered, but those objects are completely unrelated as they exist within different instances of the Alpaca Server. The extension also includes calls to the Alpaca

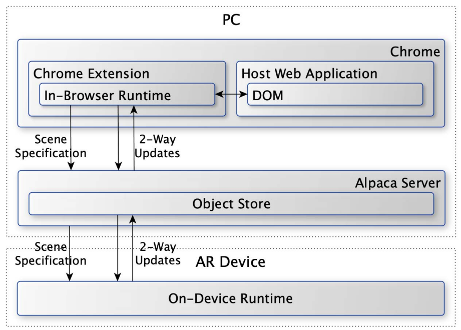

Figure 2: System architecture overview. The Alpaca Server provides the communication gateway between the web application and the On-Device Runtime and can reside on the same machine as the Chrome browser or a remote machine.

In-Browser Runtime which manages event listeners, triggered when an action is performed in AR.

Alpaca Server. The server handles all stateful information that must exist between the in-browser and on-device runtimes. It also serves as a shared information space for any content that needs to be stored for longer periods of time. The Alpaca Server is application oblivious. Its stateful information management is also agnostic of content types, regardless of persisting images, canvasses, text, or other MIME types.

One Alpaca Server can support many applications and users concurrently. For example, all applications in this work actually use the very same Alpaca Server.

Alpaca On-Device Runtime. The on-device runtime is a single application that runs on the AR device to support MR content from the Alpaca Server. This runtime performs registration and automatically refreshes the AR scene when it has been updated by the in-browser runtime. User actions that should trigger state changes on the host application are sent through the Alpaca Server.

Alpaca's on-device runtime is application oblivious. Upon application start, the on-device runtime queries the Alpaca Server for AR objects and begins to automatically set up the scene according to the specification registered by the corresponding Chrome extension. After first start, the on-device runtime will continue to operate, even if the in-browser environment is closed, because all contents sent to the AR device come from Alpaca Server.

3.2 AR Driven Design Considerations

AR helps us expand the sense of space from 2D to 3D. The physical world then becomes the user's native operational environment. Using Alpaca, we want to build a bridge between the desktop-side web world and the AR enabled human world. As a starting point, in this work we focus on extending a user's current workspace.

Applications on AR face many general and known challenges of AR [40]. The main tasks on AR are: (i) accurate registration, (ii) efficient rendering, and (iii) reliable event handling. Through extensive experiments, we found four particular dimensions that drove our design choices.

Performance. This limitation primarily comes from a need for a high frame rate to decrease user sickness. Without dedicating significant amount of resources, it is also hard to ensure uniform quality and performance across all devices. To improve performance, applications need to limit the number of polygons, and use image-based rendering wherein complex geometries are replaced by low-poly objects with textures.

Alignment. Humans can detect misalignments between virtual and physical objects easily. Better hardware can improve accuracy, just like how HoloLens uses depth cameras and the iPhone X and Google Pixel all move the alignment process closer to hardware. Application scenarios matter too. The discovery of usable surfaces is left to the OS-level software of the AR device; for instance, ARCore and ARKit use image-space techniques to detect surfaces.

Interaction. AR is not better than desktop on all interaction tasks. Some interactions are really very well suited for the desktop medium, for example, dropdown menus and precise selection. While some researchers have been designing new hardware to bring fine-tuned controls into the physical world to improve the precision of control in an AR application [10], researchers have also been improving software detection to make more usable controls [29]. In this regard, Alpaca provides a new software approach by allowing developers to easily integrate desktop and AR interactions in the same application. Alpaca integrates with the device OS for native AR interaction.

Mobility and Cost. Many AR systems are not portable due to size, weight, and system installation needs. Some high-end AR systems are not widely available due to cost as well. Alpaca is still a web-first infrastructure. It is also designed to be device agnostic.

3.3 In-Browser Runtime

The Alpaca In-Browser Runtime (IBR) needs to be able to run on any supported websites. It is made of two parts: the user-defined code and an Alpaca-provided library of support code.

In order to avoid requiring a developer to have to modify an existing web application, and also allow the in-browser runtime to be packaged and easily released, we have implemented the in-browser runtime as a Chrome extension. As a web-standard practice, each extension, when triggered, checks whether this current webpage matches a whitelist of applications that Alpaca manages. If there is a match, then the in-browser runtime is injected as JavaScript code and runs in the context of the web application.

During the setup stage, the in-browser runtime automatically attaches listeners and hooks to the web DOM objects that the developer wants to map into the AR environment. Listeners and hooks operate in pairs. They are executed when the corresponding DOM elements are created, deleted, or updated (from the server). The in-browser runtime also automatically serializes DOM objects as Alpaca Objects and creates those on the Alpaca Server through the RESTful API. The in-browser runtime also participates in creating the relevant event streams.

During the operational stage, the in-browser runtime is an asynchronous, event-based program that responds to changes in the DOM or event notifications received from the server. The concurrency management of the JavaScript functions and DOM functionalities are entirely left to the web browser. The in-browser runtime is aware of the server, but not aware of whether the scene on the AR device exists, because the user may not have elected to use an AR device. The in-browser runtime is also not aware of how the AR scene looks. Handling and rendering of the Alpaca objects on the AR device are left to the on-device runtime.

3.4 On-Device Runtime

For ease of development and feature parity between the different AR devices, the Alpaca On-Device Runtime (ODR) needs to be able to run on all supported devices, currently the iPhone X and Google Pixel. JavaScript is a common language between the two devices, which also aids in the interoperability with the Alpaca Server. Our In-Browser Runtime is written in HTML5 and JavaScript.

On the iPhone X and Google Pixel, the registration support comes from Google's WebAR project. Internally, these use OS-provided libraries on top of Google's Chrome browser.

The on-device runtime is application agnostic. It simply queries and consumes Alpaca objects stored in the on-server object store. Strict interpretation and laying out of the scene, and rendering of the objects in the scene are all controlled by the on-device runtime, however. The same as in the in-browser runtime, all on-device objects have listeners and hooks attached to each object as well.

Rendering within the on-device runtime is quite efficient because of our design choice to prioritize the use of image-based rendering on the AR device. Each object appears as a simple texture mapped surface. The actual content of the image to be used as the texture is saved on the Alpaca Server in the object store. The rendering functions of the on-device runtime use THREE.ar.js for scene registration and THREE.js for rendering.

Image-based objects can be placed "on-surface," for example to mimic the appearance of being laid flat on top of a desk surface. The image-based object can also be placed directly facing the viewer, in that case, by interactively updating the image depending on the view angle change, the user can see a responsive 3D object in the AR scene. By limiting object complexities, we are able to further optimize the rendering performance.

3.5 Non-Invasive Event Handling

Having described the runtimes, let us now discuss the Alpaca Server from an operational perspective. During initialization (i.e. when a user invokes the AR extension for the first time), the Alpaca Server holds an empty scene. The application developer updates the scene through code in the browser extension based on the DOM element state of the application. To react to events from the On-Device Runtime, the developer can add listeners using the Alpaca API.

There are three kinds of listeners in the extension: the DOM listeners update based on the changing state of the host application; the event listeners update when the user interacts with the AR device; and the scene listeners allow the AR device to react to new scene graphs. From a developer point of view, these are grouped as DOM/Event listeners vs Scene/Event listeners.

The application developer writes code similar to Figure 3 (for Google Books). In (1), the extension creates a DOM listener that will react to changes in the application state. Each time the page updates, (2) the developer needs to get the latest images of the pages of the book. For each page, (3) we want to react to press events from the On-Device Runtime by changing the current page in the web application. Finally, once the scene graph is complete, (4) we update the scene on the AR device.

After initialization, a scene should have been created. The developer can then set event listeners (e.g. press events) onto objects of this scene. Once these are set, the scene is pushed to the Alpaca Server. This process is illustrated in Figure 4 and is repeated each time the DOM listeners are triggered. We recreate the entire scene from scratch because we want to be able to handle highly dynamic DOM changes in a reliable way.

4 AR Extension Design

As coined in [50], "an interaction is an action by a user with an intent to change the state of an application". Among *select*, *explore*, *reconfigure*, *encode*, *abstract/elaborate*, *filter*, and *connect*, some of these interactions are better suited as spatial-oriented while others are better as surface-oriented. The best fit of the interaction techniques on each platform will be task dependent.

Abstract tasks [12] that users handle on-device or on-desktop can also vary. Search tasks such as *lookup* may be better suited on desktop, and *browse* better suited for AR. Search tasks such as *locate* and *explore* could fit on both environments. Query tasks such as *identify*, *compare*, and *summarize* can fit on both environments.

Alpaca's interaction mechanism allows application developers to make flexible design choices on interaction. All application code is in JavaScript, which is easy to prototype and iterate. All the application code runs on-desktop, in the typical web application setting. This is the most native programming environment for many of today's application developers [1].

The application code is triggered on first load, its main function is to create the AR scene, register the scene with the server, together with all the objects. The application developer does not have to worry about setting up listeners, hooks, or event streams. The desktop runtime will set those up automatically. Each object is then annotated as to whether it responds to different kinds of interaction events.

```
Alpaca.listenDOM(domElement, function() {  // (1)
  let domImages = domElement.getImages();  // (2)
  let ARScene = new Group();
  for (domImage of domImages) {
    let texture = loadTexture(image.src);
    let object = new TextureMappedPlane(texture);
    object.position = new Vector3(x, y, z);
    Alpaca.listenEvent("press", googleBooksNextPage); // (3)
    ARScene.add(object);
  }
  Alpaca.updateScene(ARScene);             // (4)
});
```

Figure 3: A simplified code snippet showing the DOM and event listener setup process of the Google Books application; the elements are images containing the contents of each page of the book. In this example, we go to the next page on press. In the real application, we have separate buttons that go to the previous and next pages.

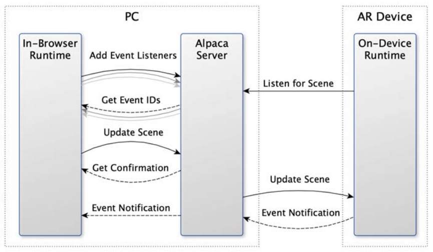

Figure 4: The scene and event listener update process when there is a DOM change in the web application, and a sample event notification flow upon user interaction. Arrow represent HTTP requests; curved arrows represent pairs of requests and responses.

Last, after the Alpaca objects have been created and properly tagged, the application code assembles the scene. The simplest way is to just add all objects into the same list. However, the application developer has the complete freedom to generate new objects, ones that are not in the DOM at all.

The assembled scene and associated objects are serialized and submitted to the Alpaca Server as one JSON object. The server will parse through the JSON, register objects individually, and store them. Scene objects are just another object of peer status as all other Alpaca objects. Again, as shown in Figure 2, the on-server object store is general and object-type agnostic and simply answers queries by the URI of the objects.

Here, we show three application scenarios using Alpaca. The first application extends a common website used by many people (Google Books), and expands the user's workspace to include books in the real world (Section 4.1). The second example looks at an educational webpage with content that can inherently benefit from 3D exposure (Section 4.2.1). The last example illustrates the creation of an AR extension when the developer has complete control over the application source and server (Section 4.3). Performance results for the applications are detailed in Section 5. *Footage of the user interactions are in the supplemental video.*

4.1 Google Books

Google Books is a large collection of digital books, which has scanned over 25 million books [19]. Using Google Books, having multiple books open and scanning through them is now a process of switching between browser tabs and viewing one at a time, rather than having the actual physical books open on a desk.

The Google Books Alpaca application augments Google Books' resources with a digital, spatially-extended workspace. The user can continue to use the web interface for doing the initial research, but they can also pull out snippets or entire books to sit around them in their workspace for easier referencing.

4.1.1 Book to Plane Mapping

Google Books renders books using images of each page as simple HTML <img> tags. The user-written script takes these images and uses them as textures for THREE.js plane objects. It then uses Alpaca's `updateScene` function to send the object to AR.

Google Books server restricts the access of their images to users on their website (e.g. via the browser). Due to this reason, the Alpaca application must explicitly upload the content to the Alpaca Server; more data than just the new scene and interactions. Hence, this application is considered *asset-heavy*.

Figure 5-Left shows a user viewing two books in AR through their phone, while making notes on their laptop. Figure 5-Right shows a closeup view of one of the books in front of the user within

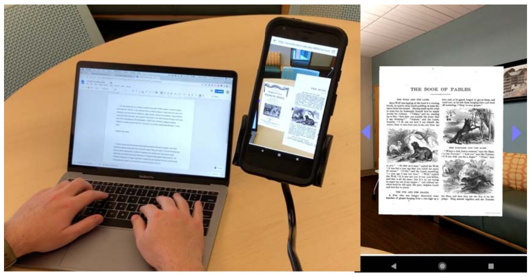

Figure 5: (Left) The 3rd person view of the Google Books application shows how the user can interact with their laptop and take notes based on the books that they have open in their AR workspace. (Right) The close-up view of the AR device shows the level of detail capable on the Google Pixel.

AR. The arrow-shaped buttons on the left and right of the book let the user go to the previous and next pages respectively.

4.1.2 Interacting with Books in AR

For interacting with the books in AR, the script creates two 3D arrow shapes using THREE.js and uses Alpaca's `addEventListener` function to add a click event for the arrows. In the callback function provided to `addEventListener`, it triggers the Google Books code as if the user switched to the next page in the browser.

4.2 Exploration of Biodiversity Through Maps

Maps are abundant on the web, and increasingly help people navigate and better understand their surroundings at different levels of granularity. In most cases, the maps are projections of a 3D world on to a 2D plane. In other words, they inherently contain 3D information. With a simple mapping provided by the developer, a 2D map on a website can be viewed in 3D in AR on the user's desk. For this use case, we picked the Species Mapper web application of the National Park Service, which visualizes and compares species distributions across the Great Smoky Mountains National Park.

4.2.1 Elevation to Height Mapping

The *Species Mapper* web application renders a map of the Smoky Mountains and provides panning and zooming capabilities. Additionally, users can view the distribution of different species rendered as an overlay on the map. Intenrally, the web application uses *leaflet.js*, a de facto library for geo-map based applications. Leaflet renders maps using tiles of images within the DOM.

We start by parsing the DOM for any underlying maps, then create a THREE.JS plane object with two textures. One texture is used for the picture of the map, another is used to encode elevation. The information from these are both gathered from Species Mapper. To make the map 3D, the user script uses a vertex-shader to map elevation to height. Finally, the THREE.js object is given to Alpaca's `updateScene` function that renders the AR version of the scene on the connected AR device. The result is shown in Figure 6.

The Species Mapper application is *scene-heavy*, because the vertex shader used for height mapping introduces more complexity to manage the scene and requires more complex usage of the AR device's GPU. Even though such complexity may hamper this application's portability to a wider range of devices, that complexity has enhanced user experience rather significantly.

The map images of Species Mapper are cloud-hosted and easily usable by the AR device. This is because Alpaca also uses URIs as the universal identifier of assets as well. In other words, the images are not uploaded to the Alpaca Server for hosting and only the URIs are sent to be fetched by the AR device through the Internet.

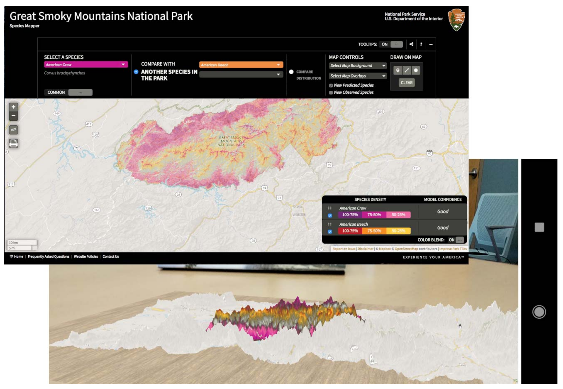

Figure 6: The AR view of *Species Mapper*. The elevation of the mountains is mapped to 3D height by the DOM-Scene mapper.

4.2.2 Map Interactions

After the user selects a species from the host web application, the desktop runtime is notified that the DOM has changed and it begins to update the THREE.js object based on the map. The AR device then updates the scene. Users then interact with the model.

4.3 Exploration of Online Recommendations

There are many web applications that include a powerful online recommendation engine. YouTube is a prime example of that. Here we develop a new YouTube-based application using Alpaca that allows users to interact with and explore in both the 2D screen and 3D physical space.

In today's web apps, recommendation is usually limited to a single 1-dimensional list of suggestions. A user can select one item from the list, upon the selection, a new 1D list of recommendations are computed based on the selection and delivered to the user. At any particular time, the user sees one list of recommendations. In a large part, this is due to the limited space on a 2D screen, and that selecting from a 1D list is intuitive and direct.

We added some automation to the above process using YouTube's public API. Specifically, after retrieving the recommendation list for one video, for each video on the list, we use a script to automatically retrieve the recommendation lists for those videos. This process recurses k-levels until a threshold number of videos are retrieved.

The process creates a directed graph. This graph is not a tree, because the same video may appear in the recommendation lists obtained during different levels of recursion. Each node in the graph is a video. To be performant in rendering, typical graphs are capped at 100 videos which equates to $k = 2$ levels of recursion when accounting for 10 related videos fetched from YouTube per video. Edges between videos indicate what YouTube's recommendation engine considers as related: i.e. a connection from A$\rightarrow$B means that B is in the list of videos related to A.

Our script automatically presents the video recommendation graph in the DOM. Using Alpaca, we can easily make the graph appear on the AR device in 3D. The AR view is comprised of the set of nodes in the graph, represented by their video thumbnail images. These thumbnails are accompanied by interactive buttons that either "like" or "dislike" the video, which are then used by Support Vector Machines (SVM) to refine the presentation of the graph.

Figure 7 shows a typical workflow for the mixed-space YouTube application. First the user starts the desktop application and is prompted to use their AR device. The user can now intuitively switch between global and local views of the suggested videos. For better visibility and exploration, the AR runtime will automatically orient the thumbnails so they always face the user.

A user can interact with the AR video recommendation graph by simply tapping on a video, the desktop in-browser view updates in a

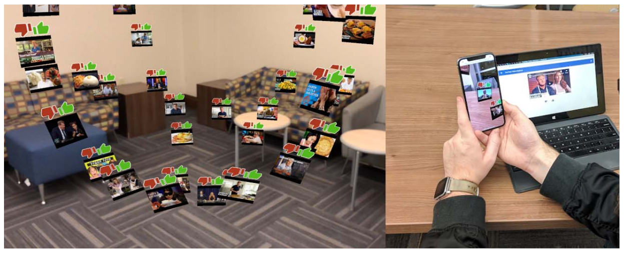
Figure 7: (left) The AR view of the recommended videos layout shows how different videos fall into distinct groups. (right) The third person view shows what the user sees on their AR device and on their laptop.

synchronized way. Event listeners are transparently attached to both the videos in the DOM and the videos in the AR scene graph. These listeners allow the user to make changes in either the desktop or AR views and have these changes affect the application as a whole.

Due to the number of videos available to the application and the difficulty in navigating such a graph, an SVM aids the user in filtering the collection of videos. This filtering occurs by the user selecting videos they "like" or "dislike" through in-device tap interactions. After a few selections, the user can trigger a classify function, upon which we train and test the SVM with the tags of the videos. This acts as a rudimentary way to increase the dimensionality of each video and improve the quality of our predictions. The filtering all happens locally in the browser.

This application is *update-heavy* because the scene graph has to be rebuilt every time the user filters the graph or watches a video and hence triggers the recommendation list to be refreshed. These updates and the resulting reconstruction process could become a bottleneck, strain the AR device, and hamper rendering rates.

5 Performance Results

Our demo apps also serve as test cases so that we can better understand the performance characteristics to bridge web with AR. In all tests, we co-locate the Alpaca Server and the browser on the same PC — a ThinkPad T420 laptop with a 2.80GHz Intel Core i7-2640M CPU and 8GB of RAM. We focus on the following metrics.

AR startup time should be $< 5s$ from when the desktop application creates the scene to when it's visible on the mobile device. This roughly corresponds to the time it takes for a user to pick up their mobile device after using their mouse or keyboard to interact with the desktop application.

AR frame rate should be in the $30-60$ FPS range to be usable. We are not directly in control of this metric because it is directly tied to the performance of the AR device, however due to rendering on the AR device instead of a remote-rendering technique, we find that this frame rate is achievable. It is worth noting that the frame rate is affected by the rendering time as well as overheads due to AR registration and sensing. Thus, AR frame rate can only improve so much before we reach the limitations of the AR device.

Server memory footprint should be <100 MB to limit the effect our system has on other services on the user's laptop. This range is around that of a normal web app and can be seen as unintrusive.

Total extension code should be around $100-1000$ lines, with around 50 lines of Alpaca application code. The Alpaca system is reusable and complex, but the application code using the API is kept simple and in only a few additional lines of code. Most THREE.js applications are already around the 1000 line mark, so adding 50 lines of Alpaca code is relatively minimal.

5.1 Alpaca Server Throughput

Herein, we evaluate data transfer rates which directly impacts the speed that the desktop extension can update the scene graph. On average, a scene is on the order of kilobytes, but the assets may total 10s of megabytes. This means that supporting 100s of MB/second should be sufficient for Alpaca applications.

Additionally, there are two test cases of importance: serial and parallel uploading. Serial uploading is where each MB is uploaded after the previous one completes. In parallel uploading, we send 4 different objects at the same time, like a browser would do in an asset-heavy application.

For the test, we repeatedly uploaded blocks of 1 MB (around the size of a single image) and measured the total time required to upload all this data. Then, we divide the total data size by the time to get a rate in MB/s. We repeated this trial in serial and parallel mode. Results are shown in Figure 8.

Due to the asynchronous way the server was written, we see a higher upload rate during parallel uploading than in serial. We suspect that the much higher increase for parallel uploading of 512 MB is due to matching the operating system's buffer sizes without exhausting lower level caches. Overall, we see that the Alpaca Server is able to handle a sustained upload rate of 400 MB/s which is more than enough for our applications.

5.2 Application Sample Runs

In this section, we test whether the Alpaca server is general and efficient enough to support multiple applications that use it, regardless of whether those applications are asset-, update-, or scene-heavy. We performed a series of sample runs for the Species Mapper and Google Books apps. In those two apps, the AR objects, and the related geometries, are auto-extracted from the existing web applications. The YouTube example app is not ideal for use in this performance testing, because that AR app required us to programmatically crawl Youtube recommendation listings and then custom create the AR objects after the multi-step crawl. The resulting runtime overhead includes multiple rounds of YouTube API queries, which is more complex. We collected metrics on: the render time for each frame of the AR device, the CPU usage by the server, and the memory usage by the server. We evaluated all tests with an iPhone X running iOS 11.1.2 to eliminate as much AR device latency as possible to evaluate the server more accurately.

The render time is measured in milliseconds and measures the time for each frame from when the rendering started to when it finished. It experiences peaks and troughs based on what the AR device is doing at the time. For example, when processing a new scene, the AR device is expected to have a longer render time and consequently a lower frame rate. We opt to show results in frames per second (FPS), instead of rendering time.

The CPU usage is measured in terms of percents of a core, as measured by the `ps` utility on Linux. This percentage can be greater than 100% when it uses multiple cores; however, because of the single threaded nature of the Alpaca Server, we find that this is never

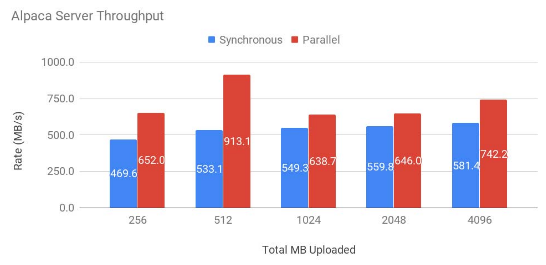

Figure 8: Server throughput as measured in synchronous and parallel modes. The data shows the maximum upload rate we can expect of the system. The synchronous mode corresponds to scene- and update-heavy applications, while the parallel mode relates to asset-heavy applications. These are differentiated by the manner of uploading, between those that must happen in sequence and those that can be concurrent.

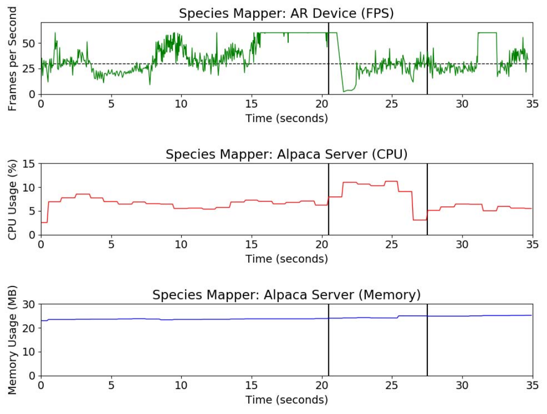

Figure 9: Alpaca Server and On-Device Runtime performance for the Species Mapper app. This is an example of a scene-heavy application that requires more compute resources to render the AR application, which lends to lower-than-average frame rates due to the AR device used. At the first mark in the graph, the user selected another species layer. The effect of this change is finished around the second mark.

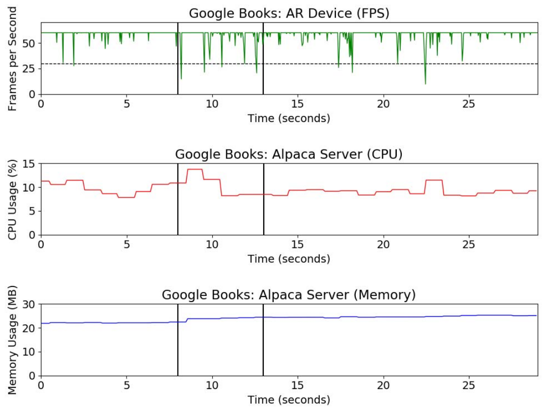

Figure 10: Alpaca Server and On-Device Runtime performance for the Google Books app. This is an asset-heavy application and must upload assets directly to the Alpaca Server to get around access restrictions imposed by Google Books. Between the two marks on the graph, the user was scrolling through the book, triggering multiple scene updates, and consequently, multiple asset uploads.

the case with our applications.

Memory footprint is measured in MB and corresponds to the virtual memory usage reported by `ps`. The Alpaca Server will use more memory as more objects are added into the object store, though we expect and find that this increase is proportional with the amount of data it is managing. The results are in Figures 9 and 10.

Figure 9 shows data from the Species Mapper app, which is scene-heavy. The render rate is between 10-30 FPS. Between the two marks on the chart (Figure 9), the user added a species layer and triggered a scene update on the AR device.

Figure 10 shows the Google Books sample run, which is an asset-heavy application. We can see that it demands higher CPU and memory usage by the Alpaca Server to handle these assets. Between the two marks on the chart (Figure 10), the user was scrolling through different pages of the book, thus triggering multiple updates to the assets in the Alpaca Server. Once this interaction has completed, the Alpaca Server's usage drops back to original levels until the user interacts with the desktop runtime again.

A summary of frame rates is shown in Table 1 (Left). Although a proper cross-device performance analysis is out of scope for this paper, rudimentary testing of these applications show that the Google Pixel can achieve an AR frame rate of $10-30$ FPS consistently, while the iPhone X is able to consistently output at $30-60$ FPS. We suspect that a major reason for this difference is that the iPhone X performs AR registration and sensing closer to the hardware.

5.3 Code Size Evaluation

As one of Alpaca's key goals, we intend that developers can make use of our infrastructure in only a few lines of code, the majority of which is typical THREE.js code. To this end, we measured the size of each of our applications (shown in Table 1 (Right)).

Table 1: The AR frame rate and lines of code of each application. The frame rate is unsteady during scene changes but then levels out afterwards. In all cases, the total lines of code is less than 1000 and the Alpaca contribution is less than 50 lines of code, under 15% of the total lines of code.

	FPS	Code Ratio
Google Books	$16-60$	26/198 (13.8%)
Species Mapper	$4-60$	17/397 (4.28%)
YouTube	$30-60$	18/466 (3.86%)

We consider the total lines of code as the number of lines, including comments and empty lines, in the extension JavaScript file. Of those lines, we consider the "Alpaca related" lines as those that directly interact with the Alpaca API (i.e. calls to `listenDOM`, `listenEvent`, `updateScene`, and functions that get passed to event listeners). We find that each of the applications is in the range of typical THREE.js applications in terms of size, and that the percentage of Alpaca related lines is below 15% in each case.

6 Discussion and Conclusion

Transforming a web application into a multi-device mixed-space application can be costly. We propose Alpaca as a reusable open-source software platform to greatly reduce that cost and avoid requiring modifying the original web applications.

Using Alpaca, any 3rd-party web developer can efficiently add an AR component to a web-application, bridge the DOM of the web and the SceneGraph of AR, and help users explore the web-application's content in a multi-device mixed-space manner. Mixing 2D screen space and 3D physical space allows a new flexibility for users to experience contents that are so far limited to web browsers.

Hence, app developers can much more broadly expand the kinds of content that can be available on AR, while reducing development-time as well as run-time costs. In turn, novel, spatial-oriented AR interactions that were primarily used for game development can now be used for and integrated with the web.

Limitations. In terms of limitations, Alpaca's use of THREE.js formats on the AR device is trading performance for reducing the workload of developers during rapid prototyping. In addition, while Alpaca expedites the development process, a developer still needs to have reasonable knowledge about both web and AR programming. As future work, Alpaca's single-user multi-device use case can be extended to multi-user multi-device settings. More extensive cross-device benchmarking of Alpaca can be valuable as well.

Acknowledgments

The authors are supported in part by NSF Award CNS-1629890, Intel Parallel Computing Center (IPCC) at the Joint Institute of Computational Science of University of Tennessee, and the Engineering Research Center Program of the National Science Foundation and the Department of Energy under NSF Award Number EEC-1041877.

References

[1] Usage Statistics of JavaScript as Client-Side Programming Language on Websites. https://w3techs.com/technologies/details/cp-javascript. Accessed: 2020-02-02.

[2] W3C Document Object Model. https://www.w3.org/DOM/. Accessed: 2020-02-02.

[3] S. Ahn, H. Ko, and B. Yoo. Webizing mobile augmented reality content. *New Review of Hypermedia and Multimedia*, 20(1):79–100, 2014.

[4] C. Andrews, A. Endert, and C. North. Space to think: Large high-resolution displays for sensemaking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pp. 55–64. New York, NY, USA, 2010. doi: 10.1145/1753326.1753336

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

[6] B. Bach, R. Sicat, J. Beyer, M. Cordeil, and H. Pfister. The Hologram in My Hand: How Effective is Interactive Exploration of 3D Visualizations in Immersive Tangible Augmented Reality? *IEEE Transactions on Visualization and Computer Graphics*, 2017. doi: 10.1109/TVCG.2017.2745941

[7] S. K. Badam and N. Elmqvist. Polychrome: A cross-device framework for collaborative web visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, pp. 109–118. ACM, 2014.

[8] S. K. Badam, E. Fisher, and N. Elmqvist. Munin: A peer-to-peer middleware for ubiquitous analytics and visualization spaces. *IEEE Transactions on Visualization and Computer Graphics*, 21(2):215–228, Feb 2015. doi: 10.1109/TVCG.2014.2337337

[9] D. Belcher, M. Billinghurst, S. E. Hayes, and R. Stiles. Using augmented reality for visualizing complex graphs in three dimensions. *Proceedings - 2nd IEEE and ACM International Symposium on Mixed and Augmented Reality, ISMAR 2003*, pp. 84–93, 2003. doi: 10.1109/ISMAR.2003.1240691

[10] M. Billinghurst, H. Kato, and I. Poupyrev. The magicbook-moving seamlessly between reality and virtuality. *IEEE Computer Graphics and applications*, 21(3):6–8, 2001.

[11] M. Billinghurst, H. Kato, and I. Poupyrev. The magicbook-A Transitional AR Interace. *Computer Graphics and Applications*, 21(3):6–8, 2001.

[12] M. Brehmer, M. Sedlmair, S. Ingram, and T. Munzner. Visualizing dimensionally-reduced data: Interviews with analysts and a characterization of task sequences. In *Proceedings of the Fifth Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization*, BELIV '14, pp. 1–8. ACM, New York, NY, USA, 2014. doi: 10.1145/2669557.2669559

[13] M. Corporation. Microsoft hololens. https://www.microsoft.com/en-us/Hololens. Accessed: 2017-10-27.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] J. Etienne. Ar.js. https://github.com/jeromeetienne/AR.js. Accessed: 2018-08-15.

[16] N. Ghouaiel, J.-M. Cieutat, and J.-P. Jessel. Adaptive augmented reality: plasticity of augmentations. In *Proceedings of the 2014 Virtual Reality International Conference*, p. 10. ACM, 2014.

[17] J. Grubert, T. Langlotz, S. Zollmann, and H. Regenbrecht. Towards pervasive augmented reality: Context-awareness in augmented reality. *IEEE transactions on visualization and computer graphics*, 23(6):1706–1724, 2016.

[18] T. Hampel, T. Bopp, and R. Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06. ACM, New York, NY, USA, 2006. doi: 10.1145/1230040.1230058

[19] S. Heyman. Google books article on the new york times. https://www.nytimes.com/2015/10/29/arts/international/google-books-a-complex-and-controversial-experiment.html. Accessed: 2018-08-15.

[20] A. Hill, B. MacIntyre, M. Gandy, B. Davidson, and H. Rouzati. Kharma: An open kml/html architecture for mobile augmented reality applications. In *2010 IEEE International Symposium on Mixed and Augmented Reality*, pp. 233–234. IEEE, 2010.

[21] H.-M. Huang, U. Rauch, and S.-S. Liaw. Investigating learners' attitudes toward virtual reality learning environments: Based on a constructivist approach. *Computers & Education*, 55(3):1171–1182, 2010.

[22] A. Inc. Apple ARKit for iOS. https://developer.apple.com/arkit/. Accessed: 2017-10-27.

[23] P. Inc. Vuforia engine. https://developer.vuforia.com/. Accessed: 2019-06-06.

[24] S. Irawati, S. Hong, J. Kim, and H. Ko. 3D Edutainment Environment : Learning Physics through VR / AR Experiences. *Science And Technology*, pp. 21–24, 2008. doi: 10.1145/1501750.1501755

[25] N. Kawai, T. Sato, Y. Nakashima, and N. Yokoya. Augmented Reality Marker Hiding with Texture Deformation. *IEEE Transactions on Visualization and Computer Graphics*, 23(10):2288–2300, 2017. doi: 10.1109/TVCG.2016.2617325

[26] R. Kooper and B. MacIntyre. Browsing the real-world wide web: Maintaining awareness of virtual information in an ar information space. *International Journal of Human-Computer Interaction*, 16(3):425–446, 2003.

[27] M. Krichenbauer, G. Yamamoto, T. Taketomi, C. Sandor, and H. Kato. Augmented Reality vs Virtual Reality for 3D Object Manipulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(8):1–1, 2017. doi: 10.1109/TVCG.2017.2658570

[28] T. Langlotz, T. Nguyen, D. Schmalstieg, and R. Grasset. Next-generation augmented reality browsers: rich, seamless, and adaptive. *Proceedings of the IEEE*, 102(2):155–169, 2014.

[29] G. A. Lee, U. Yang, Y. Kim, D. Jo, K.-H. Kim, J. H. Kim, and J. S. Choi. Freeze-set-go interaction method for handheld mobile augmented reality environments. In *Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology*, pp. 143–146. ACM, 2009.

[30] G. LLC. Google ARCore. https://developers.google.com/ar/. Accessed: 2017-10-27.

[31] B. MacIntyre, A. Hill, H. Rouzati, M. Gandy, and B. Davidson. The argon ar web browser and standards-based ar application environment. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pp. 65–74. IEEE, 2011.

[32] T. Mahmood, E. Butler, N. Davis, J. Huang, and A. Lu. Building multiple coordinated spaces for effective immersive analytics through distributed cognition. In *Proc. of Intl. Symp. on Big Data Visual and Immersive Analytics*, BDVA'18, 2018.

[33] T. Mahmood, E. Butler, N. Davis, J. Huang, and A. Lu. Improving information sharing and collaborative analysis for remote geospatial visualization using mixed reality. In *Proc. of Intl. Symp. on Mixed and Augmented Reality*, ISMAR'19, 2019.

[34] N. Marquardt, K. Hinckley, and S. Greenberg. Cross-device interaction via micro-mobility and f-formations. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pp. 13–22. ACM, 2012.

[35] E. Mendez, G. Schall, S. Havemann, S. Junghanns, and D. Schmalstieg. Generating 3d models of subsurface infrastructure through transcoding of geo-database. *IEEE computer graphics and applications*, 28(3):48–57, 2008.

[36] S. Minocha and C. L. Hardy. Designing navigation and wayfinding in 3d virtual learning spaces. In *Proceedings of the 23rd Australian Computer-Human Interaction Conference*, pp. 211–220. ACM, 2011.

[37] N. Murray. Contextual interaction support in 3d worlds. In *Distributed Simulation and Real Time Applications (DS-RT), 2011 IEEE/ACM 15th International Symposium on*, pp. 58–63. IEEE, 2011.

[38] S. Oviatt and P. Cohen. Multimodal interfaces that process what comes naturally. *Communications of the ACM*, 43:45–53, 2000.

[39] S. Oviatt, R. Coulston, and R. Lunsford. When do we interact multimodally?: Cognitive load and multimodal communication patterns. In *Proc. of the 6th Intl Conf. on Multimodal Interfaces*, ICMI '04, pp. 129–136, 2004.

[40] I. Rabbi and S. Ullah. A survey on augmented reality challenges and tracking. *Acta graphica: znanstveni časopis za tiskarstvo i grafičke komunikacije*, 24(1-2):29–46, 2013.

[41] M. Raji, A. Hota, T. Hobson, and J. Huang. Scientific visualization as a microservice. *IEEE transactions on visualization and computer*

graphics, 2018.

[42] M. Raji, A. Hota, and J. Huang. Scalable web-embedded volume rendering. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 45–54. IEEE, 2017.

[43] H. Rouzati, L. Cruiz, and B. MacIntyre. Unified webgl/css scene-graph and application to ar. In *Proceedings of the 18th International Conference on 3D Web Technology*, pp. 210–210. ACM, 2013.

[44] D. Schmidt, J. Seifert, E. Rukzio, and H. Gellersen. A cross-device interaction style for mobiles and surfaces. In *Proceedings of the Designing Interactive Systems Conference*, pp. 318–327. ACM, 2012.

[45] Z. Szalavári and M. Gervautz. The personal interaction panel–a two-handed interface for augmented reality. In *Computer graphics forum*, vol. 16, pp. C335–C346. Wiley Online Library, 1997.

[46] M. Virvou and G. Katsionis. On the usability and likeability of virtual reality games for education: The case of vr-engage. *Computers & Education*, 50(1):154–178, 2008.

[47] D. Wagner, T. Pintaric, and D. Schmalstieg. The invisible train. *ACM SIGGRAPH 2004 Emerging technologies on - SIGGRAPH '04*, p. 12, 2004. doi: 10.1145/1186155.1186168

[48] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Pose tracking from natural features on mobile phones. *Proceedings - 7th IEEE International Symposium on Mixed and Augmented Reality 2008, ISMAR 2008*, pp. 125–134, 2008. doi: 10.1109/ISMAR.2008.4637338

[49] T. J. Wang. Educating avatars: on virtual worlds and pedagogical intent. *Teaching in Higher Education*, 16(6):617–628, 2011.

[50] J. S. Yi, Y. ah Kang, and J. Stasko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE transactions on visualization and computer graphics*, 13(6):1224–1231, 2007.

[51] A. P. Yu and S. T. Vuong. Mopar: A mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '05, pp. 99–104. ACM, New York, NY, USA, 2005. doi: 10.1145/1065983.1066007

[52] S. Zollmann, C. Poglitsch, and J. Ventura. Visgis: Dynamic situated visualization for geographic information systems. In *2016 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pp. 1–6. IEEE, 2016.

[53] S. Zollmann, G. Schall, S. Junghanns, and G. Reitmayr. Comprehensible and interactive visualizations of gis data in augmented reality. In *International Symposium on Visual Computing*, pp. 675–685. Springer, 2012.