AR and VR Using the WebXR API

Learn to Create Immersive Content with WebGL, Three.js, and A-Frame

Rakesh Baruah

Apress®

AR and VR Using the WebXR API

Learn to Create Immersive Content with WebGL, Three.js, and A-Frame

Rakesh Baruah

Apress®

AR and VR Using the WebXR API

Rakesh Baruah
Brookfield, WI, USA

ISBN-13 (pbk): 978-1-4842-6317-4 ISBN-13 (electronic): 978-1-4842-6318-1
https://doi.org/10.1007/978-1-4842-6318-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6317-4. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To Mom & Dad for boundless patience, love, and support

Table of Contents

About the Author

Rakesh Baruah is a writer and creator with 15 years of experience in new media, film, and television in New York City. After completing an MFA in screenwriting and directing for film from Columbia University, Rakesh joined the writers' room of a hit, primetime, network drama as an assistant. The experience opened his eyes to the limits of television and the opportunities promised by 3D, immersive content. In 2016 he began a self-guided journey toward mixed reality design that has taken him through startups, boot camps, the Microsoft offices, and many, many hours in front of a computer. He is the author of one previous book on virtual reality and the Unity Game Engine and has received an Nvidia-certified nanodegree in Computer Vision. He currently teaches high school computer science in Milwaukee, WI. He shares what he's learned with you in a style and format designed specifically for the person who, in high school, preferred English class to Trigonometry.

About the Technical Reviewer

Yogendra Sharma is a developer with experience in architecture, design, and development of scalable and distributed applications, with a core interest in Microservices and DevOps. He is currently working as an IoT and Cloud Architect at Intelizign Engineering Services Pvt Pune. He also has hands-on experience in technologies such as AR/VR, CAD CAM, Simulation, AWS, IoT, Python, J2SE, J2EE, NodeJS, VueJs, Angular, MongoDB, and Docker. He constantly explores technical novelties, and he is open-minded and eager to learn about new technologies and frameworks. He has reviewed several books and video courses published by Packt and Apress.

Acknowledgments

Deep thanks to the members of the Immersive Web Working Group for their support of the WebXR API. To Brandon Jones, Nell, Manish, and others whom I only know through Twitter, thank you for the attention you put into the documentation for the WebXR API and all of its features. Mr. Doob, thanks go to you and your compatriots for creating and maintaining Three.js. To the team at Google Chrome Labs, thank you for evangelizing the promise of augmented reality on the Web. To Mozilla and all who have called it an employer, thank you for everything you have done to help make the Web a more inclusive, democratic space. Thank you to the team members at Mozilla Mixed Reality, Mozilla Hubs, MDN, and A-Frame for creating, supporting, and maintaining the tools to make mobile mixed reality an opportunity for everyone in the world. An incredibly special thank you to my team at Apress for their tireless devotion to my project. Spandana Chatterjee, thank you for your support and concern for all things book related and not. James Markham, thank you for the guidance you have provided for each chapter. To Yogendra Sharma, my technical editor, thank you for keen eyes and a sharp mind that kept me honest. And finally, thank you to my primary editor, Divya Modi, for whom this is my second book. Divya, thank you for the prompt responses, clarifications, follow-ups, and forwards that made collaborating remotely a smooth, fruitful experience.

Introduction

This book is a resource to help you become familiar with the tools to create mobile mixed reality for the Web. On July 24, 2020 the World Wide Web Consortium, the international standards organization for the World Wide Web, published its most recent version, as of this writing, of the WebXR API specification. The specification describes how Web browsers can implement support for virtual and augmented reality devices, including headsets and sensors, on the Web. The first iteration of the specification appeared in 2017 as the WebVR API. However, in 2018 the expansion of use cases for VR and AR on the Web prompted the Immersive Web Working Group—made up of contributors from Google, Microsoft, Mozilla, and elsewhere—to overhaul WebVR in favor of an API designed to embrace what the future of mixed reality may offer. By June of 2020, at least four of the leading Web browsers, including Google Chrome, Microsoft Edge, Mozilla Firefox, and Oculus Browser, provided support for the WebXR API.

As WebXR is a new, evolving specification, resources for its development are sparse. In this book I have created a pathway to help you prepare for the future of mobile, mixed reality development. By the book's end you will be familiar with the most common tools used for WebXR development today. These tools include Visual Studio Code, WebGL, Three.js, and A-Frame. Familiarity with HTML, CSS, and JavaScript is not required to benefit from the lessons in this book.

What follows is a road map for the rest of the course. Chapter 1 introduces the concepts behind the WebXR API as well as the tools you may need to begin developing mobile, immersive applications. Chapter 2 places us at the point of origin for 3D graphics on the Web, WebGL. By creating simple projects with WebGL, HTML, and JavaScript, you will

quickly learn the fundamentals of how the WebXR API works inside a browser. In Chapter 3 we remain with WebGL, as its bare-bones syntax makes clear the ins and outs of the graphics rendering pipeline that connects server, client, and GPU. Chapter 4 builds on the preceding two chapters, culminating with an explanation of linear algebra through WebGL. The simple, yet important, principles of linear algebra covered in Chapter 4 provide the suggested groundwork for a deep dive into immersive Web development with the 3D JavaScript library, Three.js, in Chapter 5. With a thorough understanding of the WebGL pipeline and the convenience created by the Three.js library, you will create a virtual reality project on your local machine and load it into a VR-capable device through the Internet via the WebXR API in Chapter 6. Chapter 7 moves the focus from virtual reality to augmented reality programming with Three.js. Using the features of the WebXR API's Augmented Reality module, Chapter 7 provides steps toward creating mobile AR experiences that include animation and user interaction. Chapter 8 returns to the topic of virtual reality to introduce the use of A-Frame, a framework for creating mobile XR experiences using Three.js. Both Chapters 9 and 10 remain with A-Frame, as Chapter 9 explains how to implement real-world physics and user interaction in a VR scene through the WebXR API's implementation of the Gamepad API, also built into many browsers. Finally, Chapter 10 provides instruction on how to import 3D models into A-Frame, animate them, and view them in augmented reality through GitHub Pages.

The WebXR API is poised to become a useful tool for XR and Web developers alike. As the lines between mobile and native, augmented and virtual blur, applications that make use of both 2D and immersive technologies will become more common. I have created the lessons inside this book with the intent to help you join the growing community of developers designing experiences for the immersive Web. No prior experience with Web development or 3D programming is assumed. As the WebXR API is such a new technology, more seasoned developers may also benefit from the instruction contained within. As the future of

Web development moves into a third dimension and the principles of game development move on to the Web, more opportunities will open up for creative minds to forge the language of the new Internet. I hope you, empowered with the lessons in this course, will be among those leading the charge.

CHAPTER 1

Getting Started

WebXR is not a programming language; it's not even a library of code we can access to create our apps. WebXR is a specification developed by the World Wide Web Consortium, W3C, a nonprofit group of industry experts who collaborate to create standard protocols across the Web. The W3C has left the implementation of the WebXR guidelines to the developers of browsers. WebXR, therefore, is nothing more than a set of rules agreed upon by industry.

Not to be confused with the WebXR specification, the WebXR API is an *implementation* of the WebXR feature set. The WebXR API serves as an interface between XR Web content and the devices on which they run. For example, the WebXR API collects data regarding the orientation of a headset and a user's pose. The WebXR API provides developers access to user data through its library of commands.

Yet, the WebXR Device API does have important limitations: it can't manage 3D data or draw anything to a screen. The WebXR API is not a rendering engine. It cannot load models, wrap them in textures, and paint them to pixels—a process known as rasterization. To rasterize 3D content in a browser, the WebXR API extends another API called WebGL.

Following an introduction to the components integral to the use of the WebXR API, we will discuss the tools we need to create XR applications of our own. The tools required for creating WebXR applications are a code editor, a local development server, a Web browser, and an XR device. Developers without access to an XR device may use the WebXR Emulator provided by browser creators like Mozilla. All of these are discussed in a later section of this chapter.

R. Baruah, *AR and VR Using the WebXR API*,
https://doi.org/10.1007/978-1-4842-6318-1_1

A thorough understanding of how the WebXR API builds upon the fundamental features of the Web browser will make understanding the tools we will use later in the course, such as the Three.js JavaScript library and the A-Frame framework, an easier process. By preparing ourselves with an understanding of the WebXR API from the ground up and a knowledge of how the tools we will use will impact the development of our WebXR apps, we will guarantee that we are best prepared to meet whatever advancements the WebXR API may release in the future.

In this chapter you will:

- Learn the origin and purpose of WebGL
- Briefly cover the role of JavaScript in the history of the Web browser
- Learn the purpose of the browser's rendering engine
- Learn the role played by buffers in XR applications
- Learn the value that graphics processing units (GPUs) offer to creating and running XR apps
- Survey the tools needed to create WebXR applications
- Cover the system requirements for the use of these tools
- Come to understand the suite of technologies used throughout this course

WebGL

WebGL is a Web graphics library available through a JavaScript API in all contemporary Web browsers. Like the WebXR API, the WebGL API also conforms to a specification. The specification for WebGL, however, is not maintained by the W3C, but by a different consortium known as the Kronos Group. Comprising over 150 leading technology companies, the

Kronos group promotes advanced Web standards for graphics, mixed reality, and machine learning applications. One among their many visual computing APIs is the OpenGL graphics standard.

The OpenGL graphics standard specifies a protocol for communication between an application and the drivers of a GPU, such as those made by Nvidia and AMD. While OpenGL is compatible across machines, platform-specific APIs like Microsoft's DirectX and Apple's Metal also exist. However, OpenGL's cross-platform applicability has made its younger cousin, OpenGL ES, a popular graphics API to implement on mobile devices. The ES in OpenGL ES stands for "embedded systems," which means the API targets small, low-power devices. As these devices cannot avail themselves of the big GPUs you can find in a desktop gaming computer, for example, they require a graphics API dedicated to their specific needs.

OpenGL ES' ability to operate on mobile devices allows WebGL to create 2D and 3D graphics in Web browsers running on stand-alone headsets and smartphones. It is the Kronos Group's specification for OpenGL ES that informs the implementation of the WebGL API. While the communication between applications and GPUs still requires the use of GLSL, the language of OpenGL's rendering and drawing commands, the WebGL API enables Web developers to blend GLSL with a language they are much more comfortable with, JavaScript. After all, JavaScript is the language of the Web, and the Web is the domain of the browser.

The Browser

The Web browser as we know it today really came of age in 1995 with the release of Netscape Navigator. Though Netscape eventually succumbed to the industry leviathan of Microsoft's Internet Explorer, its legacy continues to inform the nature of the Web. But Netscape wasn't even the first publicly used Web browser. That distinction belongs to an earlier iteration of Navigator called Mosaic. In fact, Navigator and its predecessor had been around since 1993. What, then, happened in 1995 to mark the year as a watershed moment in the browser wars?

JavaScript happened. While developing Navigator, Netscape sought a scripting language to use inside its browser. Originally, developers at Netscape wanted a programming language that embraced the object-oriented paradigm (OOP) of Java. However, the OOP nature of Java proved ill-fitting for the needs of the browser. Looking for outside help, Netscape recruited software engineer Brendan Eich to implement a version of the Scheme programming language for the browser. For better or worse, the minimalist dialect of Scheme didn't appeal to the larger community of developers who preferred Java's OOP approach to software design. Looking for a compromise, Netscape brass asked Eich to strike a balance between the structure of Java and the flexibility of Scheme. As the apocryphal story goes, Eich developed what came to be known as JavaScript over the course of just 10 days.

Eich's intent with JavaScript was to "touch the page." By any measure Eich succeeded, as JavaScript is one of the most popular programming languages used worldwide. Web developers have used JavaScript and members of its family like AJAX and JQuery for decades to create Web applications increasingly more responsive to user feedback. With the arrival of Node.js, JavaScript leapt from the front end to the server-side back end of Web development, an arena once exclusively dominated by more established languages like C and C++. JavaScript's flexibility has made it a go-to language for many developers interested in designing for the full stack. But its efficacy may not be more apparent than in the Web browser, where its extensibility allows for the creation of streaming XR content.

The browser is literally our window into the World Wide Web. One need not do more than execute the function window.onLoad() in a JavaScript file to understand what I mean. Really, though, the Web browser is less a window than a wall. It doesn't allow us to peer into the Web. Rather, it brings the Web into our homes, onto our tablets and our phones, by painting the contents of the Web onto the screens of our devices. About 60 times a second a Web browser repaints itself to create the illusion of a world that we surf with keyboard strokes and mouse clicks. The core of a

Web browser's functionality is its ability to render remote content to our screens. The source of this power is the product of one of its two main engines.

The Render Engine

Two engines make up the modern Web browser application. One is the JavaScript engine, such as Chrome's V8 engine, which manages the compilation of JavaScript code. The other is the engine of primary importance to us, at this point in our journey. That engine is the one responsible for rendering content delivered from a server to our screens.

When information arrives at our Internet-connected devices, it passes through the many protocol layers of the network specification before appearing inside our browsing window. Data leaves a server wrapped in layers of instructions that communicate to each node on the network how to route data to its target. Layer by layer is stripped away by network nodes until the data packet reaches the machine of the client who requested it.

If the header of the data packet matches what the browser expects, then the browser gets to work refitting the data to appear on our screen as it began at its source. Employing its ability to parse the packet's content, the browser builds a page from the syntax of its HTML document. While the JavaScript engine attends to the demands of the website's JavaScript modules, the browser's rendering engine digs into the layout and compositing instructions described through HTML and CSS. When the rendering engine is through laying out the elements of a page and painting them in the order they appear on the screen, we, the user of the client browser, will have barely noticed that any time has passed at all.

But how exactly does a browser understand where on our screens it should draw certain shapes or tint certain pixels? Sure, a designer has included the instruction set for a page's appearance in HTML and CSS, but what if a user scrolls? Enters a character into a form? Or presses play on a video? A browser requires a place to store in memory the content it

receives from a server to repaint to the page in case of update. The server too needs memory to hold data in queue as it waits to stream to the browser. What are these objects of memory called?

Buffers

If you've ever tapped your foot impatiently waiting for a Web page to load, then you're already familiar with the concept behind a buffer. Buffers are slots of memory included in hardware to hold information in bits. Buffers include addresses that inform pointers in software programs of the location of important data. Programs retrieve data from buffers before passing it through a thread on a processing unit to undergo operations. If the amount of data to move is greater than the volume, or capacity, of a thread, then a program's execution will lag. If the data are the bits of a YouTube video, then you're going to tap your foot as you wait for it to load.

Buffers are registers for memory allocation. They exist on processors, on hard drives, in RAM, and even virtually in the browser as cache. Much of creating XR for the Web relies on the efficient storing and retrieving of data from buffers; they are an important part of the WebGL specification. Transferring data to and from buffers can be costly and can destroy the believability of an immersive experience if causing lag. Fortunately, the rapid filling and emptying of buffers has been significantly improved by the increasing availability of desktop and mobile GPUs.

The Graphics Processing Unit

GPUs are computer chips that specialize in parallel processing. CPUs, central processing units, are the brain of computing devices. Their embedded logic gates and internal clocks are the essence of digital computing. Over time, CPUs have increased their productivity through the

inclusion of more cores. Broadly speaking, cores on a CPU align with the number of processes a chip can run at the same time. More cores mean more threads, which mean a greater capacity of the computer to execute tasks concurrently. The number of cores serves as a benchmark for the speed of a processor. Whereas higher-end CPUs can have somewhere around eight cores, consumer-grade GPUs can have anywhere from the hundreds to the thousands.

Today GPUs power much of the intensive computing required by AI applications in industries as far and wide as self-driving cars to protein synthesis. Their popularity, however, grew because of the breakthroughs made by designers of video games. Like the Web browser, video game applications paint and repaint a screen up to hundreds of times a second. Each frame update requires calculations of character positions, environment, lighting, cameras, materials, textures, and more. The faster and more detailed a game, the higher the demand on a machine's rendering power. Applications implementing the specifications of OpenGL, such as Microsoft's DirectX, leveraged the parallel processing of GPUs and their many, many cores to create video games that could compute and render complex character geometry at rates and volumes never before seen.

As the prevalence of GPUs in consumer machines has grown, so too have the availability and demand for virtual reality content. The speed at which GPUs can calculate the shape, color, position, and orientation of objects to a screen has supported the beginning of a new era in 3D graphics. Contemporary techniques for rendering through GPU computation, such as raytracing, have blurred the line between the real and virtual in ways that are equally exciting and unsettling. But the evolution of GPU tech isn't limited to beefy consoles and gaming PCs. Advancements in engineering and chip design have shrunk the power of GPUs to the nanometer scale, bringing the wonder of 3D to mobile and handheld devices.

The Present Future

Chipsets in modern mobile VR headsets and smartphones are pushing the envelope of what has been possible to achieve through computing. As the parallel execution of GPUs and newer system architectures arrive on more and smaller devices, the demands placed on machines to render XR content in real time will become less daunting. The WebXR API, by extending the WebGL API (which is itself based on the specifications of OpenGL ES), allows us as XR content creators to leverage the power of GPUs to bring virtual and augmented experiences to hundreds of millions of people through the Internet.

In designing JavaScript, Brendan Eich may have aimed to give designers the ability to touch the page of a website. Twenty-five years later JavaScript endures, and through the WebXR API in the browser, provides us, designers, with the ability to touch reality itself. In the remainder of the chapter you will learn the tools required to build XR content with the WebXR API.

Tooling Up

The tools described in the following sections have proved helpful to me during my development of WebXR content. Some are required; others are not. Each has been vetted by reputable parties if not directly by me. As always should be the case when creating with a bleeding edge technology like WebXR, refer to the most recent, published documentation for up-to-date compatibility and requirements.

A Code Editor

Like a text editor, a code editor allows you to type the syntax of a program into a document. Features built into a code editor create an

environment convenient to writing, deploying, testing, and correcting code. Throughout this book I use Microsoft's Visual Studio Code editor (VS Code). It is cross-platform, popular, powerful, and free.

We will use it to write the HTML, JavaScript, and CSS required to create XR applications for the Web. As VS Code also includes a marketplace for convenient developer extensions and integration with GitHub's version control platform, it enjoys widespread popularity among developers of all stripes.

Visual Studio Code download requirements from Microsoft's documentation are as follows.

Hardware

Visual Studio Code is a small download (<100 MB) and has a disk footprint of 200 MB. VS Code is lightweight and should easily run on today's hardware.

We recommend:

- 1.6 GHz or faster processor
- 1 GB of RAM

Platforms

VS Code has been tested on the following platforms:

- OS X Yosemite
- Windows 7 (with .NET Framework 4.5.2), 8.0, 8.1, and 10 (32-bit and 64-bit)
- Linux (Debian): Ubuntu Desktop 14.04, Debian 7
- Linux (Red Hat): Red Hat Enterprise Linux 7, CentOS 7, Fedora 23

Additional Windows Requirements

Microsoft .NET Framework 4.5.2 is required for VS Code. If you are using Windows 7, please make sure .NET Framework 4.5.2 is installed.

Additional Linux requirements

- GLIBCXX version 3.4.15 or later
- GLIBC version 2.15 or later

For a list of the most recent requirements, visit: https://code.visualstudio.com/Docs/supporting/requirements#_platforms.

Local Web Server for Development

To test and debug Web applications written into a code editor, developers require the creation of a local Web server. Mimicking the behavior of a remote server that stores and delivers Web pages and their resources to client browsers, a local Web server allows developers to launch and view Web applications from their local machines. For the exercises in this book, I use the Live Server extension created by Ritwick Dey, available for free in the VS Code Extension Store.

Live Server VS Extension by Ritwick Dey

See https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer.

Other popular options to create a local Web server are modules available through Node.js and Python. Both Node and Python require installation on your machine before providing access to their local server resources.