6-2024

Virtual Field Environments Capstone Software Review

Ashton Sawyer
Portland State University

Virtual Field Environments Capstone Software Review

by
Ashton Sawyer

An undergraduate honors thesis submitted in partial fulfillment of the

requirements for the degree of

Bachelor of Science

in

University Honors

and

Computer Science

Thesis Adviser

Brenda Glascott

Portland State University

2024

This essay describes my performed tasks and workflow during my computer science capstone project. The capstone is a two-term (six month long) class designed to provide students with real-world software development experience. Students are split into teams of six to eight members, each team working on a project proposed by sponsors—individuals or organizations—from the local community.

I begin by describing my project's requirements and the architecture that was devised to satisfy them. Then, I detail my team's development process, including the selection of technologies and tools, the workflow, and the challenges we encountered. Additionally, I examine some of the parallels between the capstone and real-world development, providing insight for future students and potential sponsors.

My capstone project was sponsored by PSU geology professor Rick Hugo. One of the classes that Hugo offers involves creating virtual field environments (VFEs), web-based apps that consist of a collection of 360º scenes with clickable "hotspots" that pop up various types of media, for K-12 students. Each VFE offers a detailed tour of a real field site, and they give students the chance to either scout a location before an in-person field trip or to access a site they would otherwise be unable to visit.

Previously the VFEs were made using Holobuilder, a proprietary software designed for construction management. Because Holobuilder was not designed for Hugo's use case, some desired features were missing or difficult to use. My team's goal was to recreate Holobuilder's base functionality and add the features that Hugo wanted.

One of the first things that the team did was agree on a list of terminology to hopefully prevent future miscommunication. A *VFE* is a collection of *photospheres/scenes* plus a *navigation map*. A photosphere is a single panorama and its collection of *hotspots*, which are

clickable markers that lead to further media. The navigation map is a 2D image populated by hotspots that lead to other photospheres. A *popover* is the component that appears to render media when a hotspot is clicked. A *nested popover/hotspot* is a popover that contains more hotspots to interact with. Figures depicting these components in both Holobuilder and our software can be found in the appendix.

In our initial planning, we decided that the project could be broken down into three main components: a viewer, an editor, and the backend remote storage and hosting. The viewer is the part of the project that ends up in front of the students. It renders the navigation map and scenes, allows navigation between scenes, and pops up the additional media when hotspots are clicked on. The editor is the part of the project that Hugo and his students would work with to create or modify VFEs. The backend for the project would involve storing and retrieving media and data structures for each VFE. We decided it made the most sense to start with the viewer and then work our way down the list, so our first task was to find a way to render the VFEs.

Before starting the project, Hugo gave our team a few suggestions for open-source starting points. Since none of us had much experience with 3D web rendering, we took his suggestions and started looking through them. The first, and initially most appealing, was Marzipano, a virtual tour software that already had many of the features that we needed. It also already had an editor. However, we ultimately decided against using it because it had such a large codebase and hadn't been updated in 3 years. The expansive codebase would have taken a long time to read through and understand to the point that we could add our own features. If we were to find it too limiting or difficult to contribute to it would have wasted a lot of time. The lack of updates meant that future support was unlikely, which might make our project difficult to maintain.

The two other starting points given by Hugo were ThreeJS and photo-sphere-viewer.js. ThreeJS is a relatively low-level 3D rendering library and photo-sphere-viewer.js is a library specifically designed for displaying panoramas as 360º spheres. We knew we wanted to work with React before we knew which library we might prefer. Fortunately, both of these libraries had React wrapper libraries available, react-three-fiber and react-photo-sphere-viewer respectively. In the end, we decided not to use react-three-fiber because it was too low-level. We would have had to build many more components by hand than we did with react-photosphere-viewer, which came with plugins to handle moving from one scene to another, interactable hotspots, and the navigation map. In fact, photo-sphere-viewer.js was written using ThreeJS.

Since we were trying to create a web application it was given that we would be working with some form of JavaScript. We decided to use React because it lets you abstract elements of the DOM into dynamic and reusable components that create a better structure than pure JS and HTML. We chose React over another framework like Angular because some members of the team were already familiar with React after taking the Front-End Web Development course offered at the university. We also decided that we wanted to use TypeScript, a strongly typed superset of JavaScript. TypeScript compiles into JavaScript, so they can be used in the same locations, but the required typing of TypeScript helps to structure and document code in addition to catching certain would-be run-time errors in the compiler.

When it came time to start coding, we tried to structure our development process well to prevent as many mistakes from reaching the main code base as possible. We used git and GitHub for version control. We used branch protections so that any commits to the main branch required a pull request, and each pull request had to be reviewed by at least one other team member

before merging. We also used GitHub Actions so that any pull request had to pass integration checks from our compiler, linter (ESLint), and formatter (Prettier) before merging.

Our software development process was incremental and Agile-like. We split up our work into 1-to-2-week sprints during which a feature was specified, implemented, and integrated into the main branch. Every Monday we would have a stand-up meeting over Discord to discuss what we had worked on the previous week and any roadblocks that may have come up. Then we would get our tasks for the next week. Other than our team lead, who was selected by the capstone instructor and had a hand in selecting the team members, no one was assigned a formal role within the team. Tasks were received each week based on interest and experience. Part of the team leader's job was to prepare a short list of tasks from the backlog to be given out for the week based on complexity and impact.

Starting from this short list, we would each volunteer for one or more tasks. At most two people were assigned to each task, and only if the task seemed large enough to warrant multiple developers. If a task wasn't assigned to anyone, it would be put back into the backlog to discuss another week. If a task from the previous week wasn't finished, then it would be reassigned to the same person or persons. If someone finished their task from the previous week then they would either be assigned a new task or help with one of the unfinished tasks. We used GitHub Issues to keep track of our backlog of tasks and added to it as necessary.

During the first term of our two-term capstone, we would also meet in person on Wednesdays to plan the project more generally. This included things like creating the data structure associated with a VFE, preparing for our mid-project presentation, and meetings with Hugo. As we got farther into the project, we found there was less need for these meetings and

stopped holding them. Instead, team members who were working on a task with a partner were expected to schedule times to meet on their own as needed.

The backend for the project was one of our main challenges and remained difficult throughout the project. Hosting was easy to do. We used Vercel, which automatically deploys the main branch of the repository to a constant URL and provides previews for each commit on other branches so that team members can view each other's work before it gets merged. We also used Vite to deploy the project on a local server during development. The problem was with the storage. We didn't originally plan to have a robust back end, but for a created VFE to be useful it must be savable, loadable, and distributable. That means storage of some kind needed to be supported.

First, we checked with the CAT to see if they could provide the storage space, but they could not. This meant that we had to look towards third-party cloud storage options. The problem with third-party providers is that they cost money. Most providers have a free tier that supports up to a certain threshold of storage and usage, often 2-5GB. While this is a fair amount of storage, we assumed that Hugo might eventually want to move his previously created VFEs into our system, and he already had 105GB worth of data. Even if we were to disregard Hugo's backlog of assets, we would only be able to store about seven new VFEs before hitting the storage limits, given that our incomplete demo VFE's assets took up 262MB. No matter the scenario, if we picked a third-party cloud storage provider, eventually Hugo would have to start paying.

Five weeks before the project deadline, we decided that we had to choose something. Each team member was assigned a different platform to research before the next meeting and present as an option to the group. The big three cloud providers–Amazon, Google, and Azure–

were very similar both in services offered and pricing. Other services like MongoDB, Firebase, and Cassandra were also considered, though we found that many of these services acted as wrappers for one or more of the big three.

We also considered loading and saving the VFE's assets and data locally, and ultimately this was the solution we chose. Rather than hosting the VFEs remotely and fetching them through a unique URL that could be given as a link to students, a teacher would create their VFE, save it to a zip file, and then email the zip file to their students. Each student would then upload their zip file to access it in the viewer. The foremost benefit of this solution was that it is completely free. Another unintended benefit is that user authentication would no longer be needed to restrict access to the editor portion of the program. If the VFE were hosted remotely, we would have to ensure that not just anyone could access the editor, otherwise it would affect every other person who attempted to access it. With local storage, if someone edits their VFE, only their personal copy is affected.

One of the reasons that we didn't move forward with local storage sooner is because of the limitations of a browser feature called *localStorage*. LocalStorage allows data stored in the browser to persist even when the browser is refreshed or closed and reopened. This is useful so that a loaded VFE doesn't disappear every time that the page is refreshed and so that VFE's don't have to be reuploaded every time the browser is closed. The problem with localStorage is that it has very limited space, and the exact amount isn't standardized. It ranges from 2-10MB, which is nowhere near big enough for a full VFE. LocalStorage is also only able to store string values, meaning that the data structure for the VFE would have to be converted back and forth between a JSON object and a string, which reduces performance. However, we were able to find a library called localForage that provides the same benefits as localStorage, but with large

enough storage capacity and the ability to save nearly any data type. Using localForage and JSZip, a library to create and read zip files, working with local storage was a viable and relatively simple option.

A personal challenge with the project was my unfamiliarity with React. I came in having taken Intro to Web Development, which provided a solid understanding of HTML, JS, and CSS, but we never worked with React. The syntax of React is something of a combination between HTML and TS and I found it relatively easy to pick up. The part that was difficult for me was how state, rendering updates, and passing information between components are handled.

One of my responsibilities was to create the form for the editor that adds hotspots to the scene and one of the pieces of information needed is the placement of the hotspot within the photosphere. While it would be acceptable for the user to simply enter the pitch and yaw as numbers on a form, it isn't ideal because pitch and yaw is a coordinate system unfamiliar to most people and would require a lot of guessing and checking to get the hotspot in the right place. A much more intuitive and user-friendly solution is for the user to be able to click on the location within the photosphere and automatically get the coordinates.

My instinct was to treat the components like classes in object-oriented programming, wherein I would be able to pass the Viewer as an argument to my component and call a getter function for the location. Instead, I had to have the editor (the closest shared parent between the AddHotspot and PhotosphereViewer components) keep track of the coordinates as a state that gets updated by a function passed to the viewer and read by arguments passed to the AddHotspot component. There are two problems with this solution. First, when a React component's state changes, it re-renders its children. This means that when a user clicks on the viewer to get the coordinates, the state in the editor changes, which re-renders the AddHotspot form. When the

form is re-rendered, its input fields are set to their default value rather than what the user had previously input. The second problem, though less impactful to the user, is that the editor is managing a piece of data that doesn't belong to it. The pitch and yaw of the to-be-added hotspot is information that the hotspot adder needs, and to have the editor managing it is poor abstraction.

Despite these issues, the component is functional. One benefit of not knowing React at the beginning of the project is that I was able to gain experience learning a new technology very quickly. Most developers won't work in the same language throughout their entire career. Rather, they will use whatever is the preferred language of the company that they are working for. If they are unfamiliar with that language, they will be expected to learn it during the training period. As such, practicing picking up new languages is very important.

The most valuable experience from the capstone was working for an extended period on a single project in a team. Many of the required computer science courses explicitly prohibit working in groups for the sake of academic integrity. Upper-division electives often allow and/or encourage some form of collaboration, but I have only taken one other computer science class wherein working as a group and sharing code was expected. No one works in isolation in the field. My teammates' code often served as a reference for me while coding, especially when I was working with parts of the language or libraries that I didn't understand well. We had to practice good communication, ensuring that we knew what everyone was working on and were able to agree about next steps for the project. We had to ask each other for help. We had to problem solve as a collective and compromise when there was disagreement.

Working on the same project over the course of two terms gave the chance to embrace iterative development through multiple sprints. When working on a project for no more than two

weeks, the goal is something that works and is good enough. There isn't time to worry about improving the user experience or the code structure and documentation. However, most jobs aren't building a project from the ground up. They require maintaining legacy codebases, updating APIs, or adding single features to already functional services. Because of the length of the capstone, we were able to practice identifying things to improve and then refine them instead of moving on once something was functional.

Overall, the capstone project was a success. We delivered a functional replacement for Holobuilder to our sponsor, meeting our primary goal. Additionally, we solidified the fundamental concepts we learned in lower-division computer science courses, like abstraction and data structures. It also allowed us to practice the soft skills that will benefit us in industry roles. We practiced critical and creative thinking while problem solving, and technical writing to prevent miscommunication. As such, this experience was a great way to conclude our degrees and will serve us as we transition into the professional world.

Works Cited

ESLint. https://eslint.org/. Accessed 22 May 2024.

GitHub. https://github.com/home. Accessed 22 May 2024.

Holobuilder, https://www.holobuilder.com/. Accessed 22 May 2024.

JSZip. https://stuk.github.io/jszip/. Accessed 22 May 2024.

Lazzari, Elia. *Elius94/React-Photo-Sphere-Viewer*. 2022. 5 May 2024. *GitHub*, https://github.com/Elius94/react-photo-sphere-viewer.

localForage. https://localforage.github.io/localForage/. Accessed 22 May 2024.

Marzipano - a 360° Viewer for the Modern Web. https://www.marzipano.net/. Accessed 22 May 2024.

MDN. *JavaScript*. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed 22 May 2024.

---. *Window: localStorage Property - Web APIs*. 8 Apr. 2023, https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.

Megaport. *AWS, Azure, and Google Cloud: The Big Three Compared*. https://www.megaport.com/blog/aws-azure-google-cloud-the-big-three-compared/. Accessed 22 May 2024.

Meta. *React*. https://react.dev/. Accessed 22 May 2024.

Portland State University. *BACHELOR'S PROGRAM IN COMPUTER SCIENCE CAPSTONE*. https://www.pdx.edu/computer-science/bachelors-program-computer-science-capstone. Accessed 22 May 2024.

Prettier. *Prettier · Opinionated Code Formatter*. https://prettier.io/index.html. Accessed 22 May 2024.

Sorel, Damien. *Photo Sphere Viewer*. https://photo-sphere-viewer.js.org/. Accessed 22 May 2024.

Three.Js – JavaScript 3D Library. https://threejs.org/. Accessed 22 May 2024.

TypeScript. https://www.typescriptlang.org/. Accessed 22 May 2024.

Vercel. https://vercel.com/home. Accessed 22 May 2024.

Vite. https://vitejs.dev. Accessed 22 May 2024.

Appendix

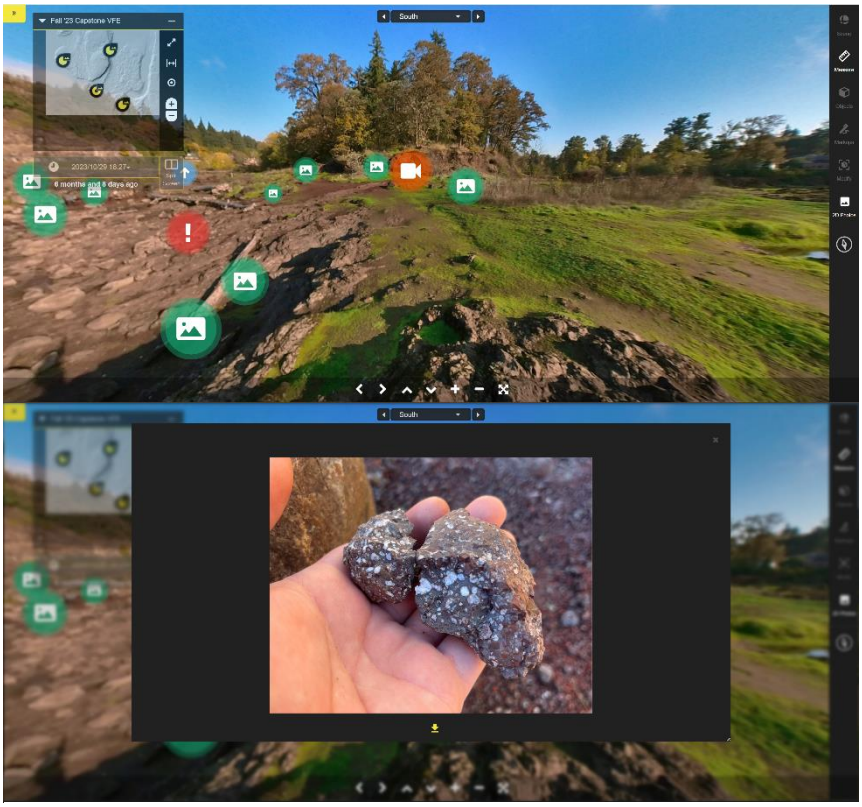

Figure A1 View of a VFE and popover in Holobuilder

Figure A2 View of a VFE and popover with a nested hotspot in our program