Unity WebGL Simulation Transition Report

PS Technology

By Logan Cooper, Beck Bolinger, Devin Gao, George Karachepone, and Carter Pasqualini

Summer 2020, 5/11/20 - 6/12/20

Table of Contents

Introduction

PS Technology is a technology firm that specializes in developing training software for the railroad industry. The company they are specifically designing their software for is Union Pacific Railroad. Their line of work consists of developing training simulations for railroad employees. These training simulations are designed to provide a better service for training conductors by integrating safety rules and procedures the conductor must follow as well as implementing features to give the trainee immediate feedback on their performance. PS Technology's simulations also cut down the cost of the normally expensive training programs that require a real locomotive train, as well as crew to keep things in check. With their modern solution, PS Technology aims to create more safe and more effective training software that will benefit Union Pacific Railroads as a whole.

This website development project was aimed at solving several technical challenges relating to transitioning existing Unity-based simulations to an internet-based WebGL system. The technical hurdles that needed to be overcome include dynamically adding new simulations to the front-end website without additional coding, creating a high score table on the website utilizing the data from the Unity simulations, dynamically loading assets from a URL into the WebGL simulations, and sending function calls between the frontend website and the WebGL backend.

The project was completed using Angular 9 to develop the frontend website, Java with the Spring libraries to develop a RESTful API for the backend, and Unity with its built-in "Compile to WebGL" option for developing the simulations. The version of Unity used in this project is the 2019.3 release version family. A number of basic sample simulations created using Unity are provided, each highlighting a specific piece of functionality. Additionally, both a simple frontend and backend were created, demonstrating the functionalities required to solve the technical challenges of this project.

As a final overview, the website that we have created consists of a stylish home page, convenient and visible dashboard that allows dynamic loading of simulations, and a game page for each simulation with a high score table and some buttons that interact with the game. Across these pages and between the four simulations provided, all the technical challenges are solved with the solutions implemented. We also created detailed documentation of how each technical challenge was solved, including all relevant information.

Requirements

The project had several functional requirements that needed to be met before the final code was delivered. These included simulations being able to be added to the website without requiring additional coding (such as by dragging & dropping into a folder) and Unity WebGL simulations being able to interface with the website via JavaScript function calls (such as to cause the game to pause or have the player jump). More functional requirements included the application needing to be hosted using the free tier of a cloud host provider, with properly playable simulations developed in Unity and exported using the WebGL export option. On a web design side, functional requirements included having a dashboard from which the user can access all simulations, and embedding the simulations into the webpage itself. Another requirement was to have the ability for the simulations to save high scores and then display these high scores into a table on the simulation page. Additionally, the clients asked that we implement a way for assets or components of the Unity simulations to be dynamically loaded at runtime. This would allow for the clients to update their simulations without the need to rebuild their Unity simulations, assuming their new assets were prebuilt with all of the needed dependencies. The dynamic loading would also allow for the general management of simulation assets to be much more efficient.

There were also several non-functional requirements to work with during the project. Since the project was more about building the frameworks to host simulations, the simulations themselves did not need to be very complicated or sophisticated. Instead just needed to act as a proof of concept for the features we needed to implement regarding simulations. However, it was important that the simulations were able to run properly and efficiently within the web pages, without causing system lag on the user's computer or web browser. The website also needed to be aesthetically pleasing and easy to understand for an end-user. We were given a set of specific tools to attempt to develop the project with, those being the free tier of Unity, a frontend in Angular and a backend written in Java using the Spring libraries. The code also needed to be refactored for quality and ease of modification, as well as being well commented and documented so that PS Technology is able to easily understand the code. Finally, the code was designed to be easy to deploy to different server architectures with minimal changes required to the codebase, beyond simply updating internal URLs.

PS Technology asked that we find a way to prioritize the loading of certain asset bundles over others in the case that there might be a folder containing asset bundles that multiple simulations will need to load. Each simulation might have its own asset bundle folder containing asset bundles of the same name, and our system should allow for these specific asset bundles in simulation specific folders to be prioritized over those with the same name in the common asset bundle folder. The final non-functional requirement of the project was that the codebase should be structured as to be easy to extend to add additional functionality after the completion of this project.

This project was primarily a proof-of-concept to determine the technical feasibility of transitioning existing standalone Unity simulations to web-accessible WebGL simulations as opposed to creating a full-scale production website. This is enabling PS Technology to make an informed decision on if they should make the transition from their desktop application based simulations to migrating them to WebGL applications hosted on a website. As such, we were able to determine much of the layout of the website and the content of the simulations ourselves, so long as they were quite simple and demonstrated the requested technical functionalities.

System Architecture

Our code and development were driven by a layered architecture diagram that was created very early on in the project, visible in figure 1. It contains each of the five layers of the project, as well as all the interactions between different layers. The layers were created using standard system architecture layers, with each component organized into the layer that it best matched.

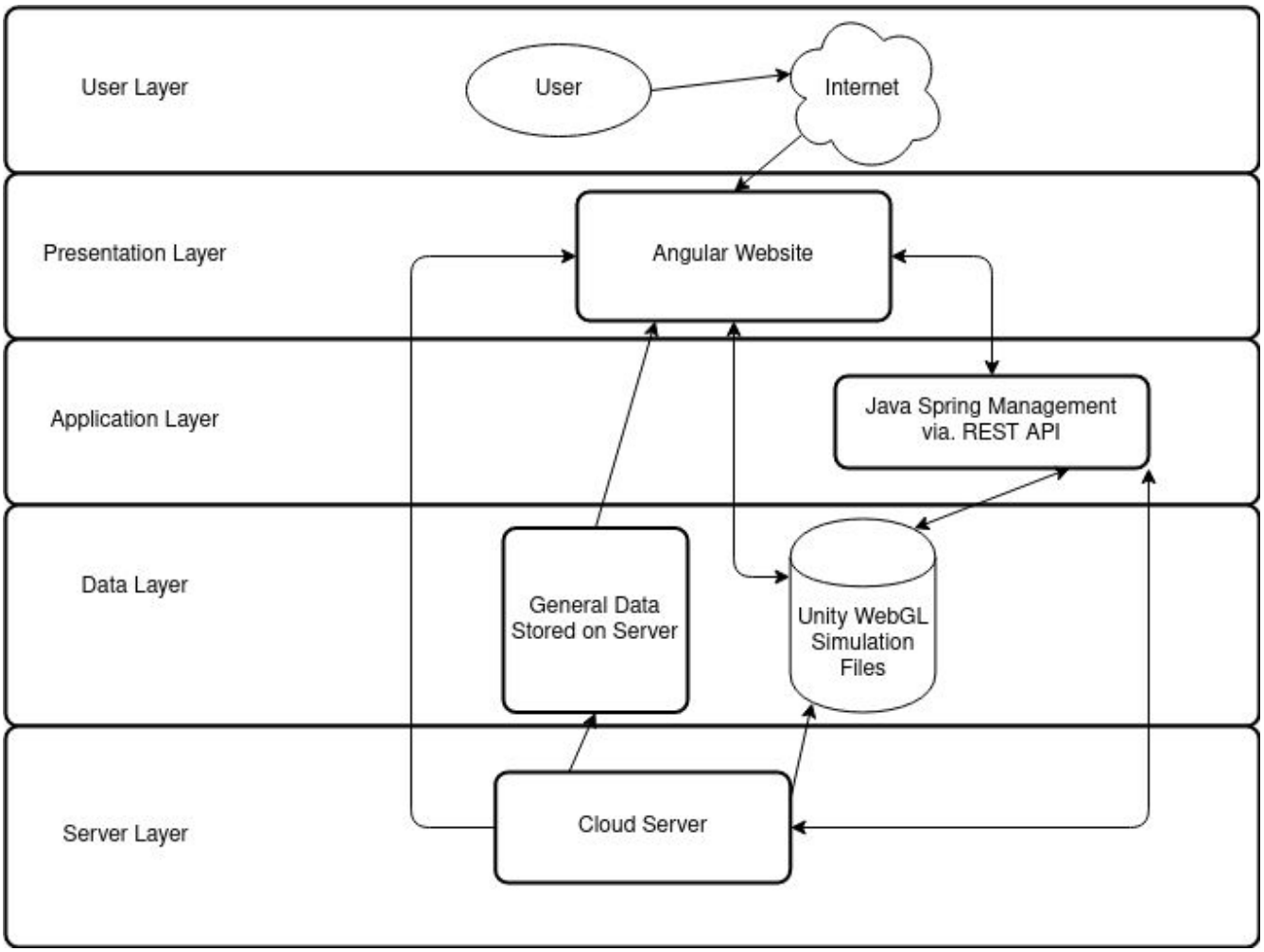

Figure 1: Layered Architecture Diagram

The User layer describes how the end-user will access the project once it is completed and deployed. Since the simulations and the framework itself are hosted on a cloud server, the way a user will access them is through an internet browser connected to the internet. The user's browser is the primary interface between the user and the website, providing a way for the user to interact with the

rest of the application. Connecting to the server directs the user to the website's home screen, bringing us to our next layer.

The Presentation layer is where the Angular frontend is located and serves as the means through which the end-user will interact with every aspect and component of the project. The Angular frontend receives various bits of information from the lower layers such as the list of simulations and also broadcasts some information back to the lower layer such as scores by subscribing to and consuming endpoints (a place to connect to the backend) provided as a RESTful API by the backend of the website. The Angular website is also what allows the user to load and interact with the various simulations that are hosted on the server, by embedding simulations into the webpage dynamically as the user selects different simulations within the application. It also plays a part in allowing there to be a user interface for simulations outside of the Unity embed itself - by providing buttons that send various commands to the Unity simulations.

The Application layer is where the Spring RESTful API is located. It handles the moving of information between the Presentation and Data layers by exposing endpoints. These endpoints are provided on port 8080 of the server running the Spring backend that the Angular website can use to get or send data. To send the list of available simulations, the API loads a designated text file containing a list of valid simulations on the web server and records the items in a JSON list (a text-based format for storing lists of data) that can be sent to Angular. Spring will also interface with the general data in the data layer through the server layer in order to send existing scores to Angular or store a newly recorded one.

The Data layer is where the bulk of the information pulled by the Presentation layer is located. This includes the filesystem where the Unity simulations and their respective assets are stored. Specifically, this would include the build of each simulation and the storage of all of the asset bundles for each individual simulation or shared between many. The layer also contains miscellaneous data and programs that are needed to make everything else work.

The Server layer involves the actual hardware and server space that everything is hosted on. The URL the end-user will enter to reach the website will link to the server and send the user to the Angular website. The server is also where user scores for each simulation are directly stored. For this project, the targeted server deployment at the start of the project was an Azure free tier server. However, due to severe computation time restrictions on the Azure free plan as well as overall time constraints of the field session, we instead decided to test and deploy the code to an OVH Cloud server instead. Both testing and deployment of the final product were done using OVH Cloud servers that have an identical technology stack to the originally targeted Azure servers. This was done to make future deployment by the client to a paid Azure server significantly easier.

Technical Design

Website to WebGL Interactions: Buttons

To demonstrate the possible interactions between the Angular website and the WebGL simulations, we were asked to incorporate buttons into our simulation pages that would be created as part of the web page, but would send messages and interact with the Unity WebGL simulations embedded into the page. This required communication between the HTML and JavaScript on the frontend of development with Angular, as well as message retrieval from within the WebGL builds and C# scripts included in these builds.

First, we developed simple HTML buttons within our Angular webpages. These buttons made function calls defined in the logic component files of the simulation pages. To make function calls to the Unity WebGL simulation instance, the HTML page needs to access the instance of the simulation by going into the iframe it is embedded into. Once the instance is accessed, we would call methods in the WebGL simulation instance utilizing the SendMessage() method that Unity is designed to understand.

The SendMessage() method calls a specific game object within a simulation and sends a message to trigger a method attached to the object, and can pass in a value into that method if it has a needed parameter. We dedicated a specific GameObject in our simulation that has scripts attached containing the methods that would produce the desired interactions with other objects within the game. For example, one of the SendMessage() calls we make within the website's JavaScript looks like SendMessage(JavaScriptHook, TogglePause()). This accesses the JavaScriptHook GameObject in the corresponding Unity simulation instance, then calls the public method, TogglePause(), from within a script attached to the JavaScriptHook, pausing or unpausing the simulation. When put together, this allows the buttons on the Angular website access the WebGL instance, call specific methods, and the WebGL instance to then execute them without significant delay. The buttons could be designed per simulation or they could be universal through all simulations, such as a pause button, assuming that the simulations have the proper elements to execute the desired function calls. As shown in figure 2, we created a pause button on a simulation web page that calls a method within the simulation itself, pausing or unpausing accordingly.

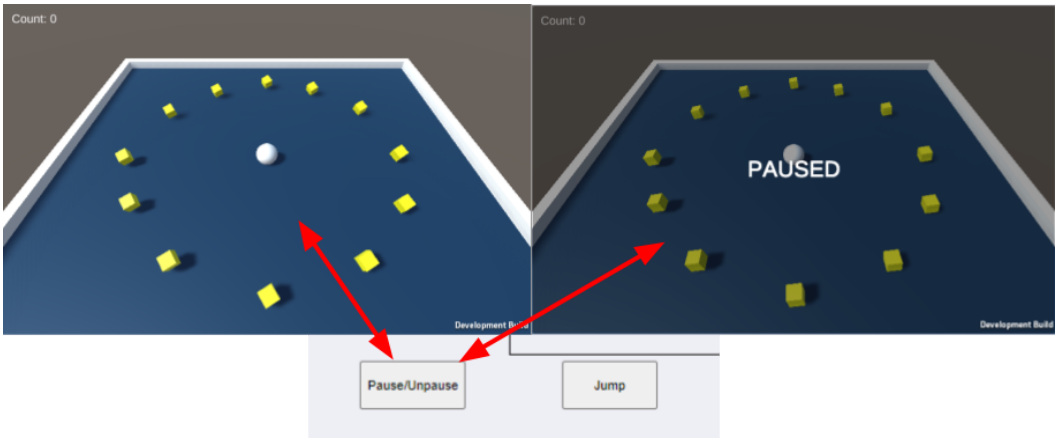

Figure 2: Pause Button Interaction

WebGL to Website Interactions: High Score Board

Simulations that have some scoring system and can utilize some sort of scoreboard was one of the examples suggested to us as a way to demonstrate interactivity between the simulations and the website. The scoreboard of a simulation is handled via the Angular frontend making calls to the RESTful API constructed in Spring. These calls include storing new high scores and retrieving those already in the system. It can retrieve every recorded score for a simulation or just the score for one user if it exists. The scores are sorted in descending order by Angular and then displayed on the website by using ng directives in the simulation page's HTML component. Certain simulations have the functionality to send scores to the database so this component really ties every aspect of the project together. This overall data flow is outlined in figure 3.

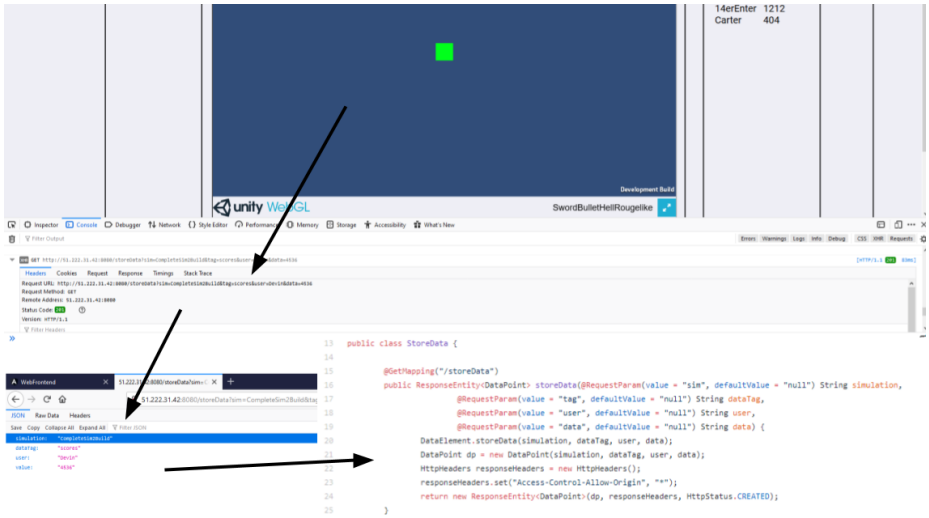

Figure 3: Data Flow from Backend to Frontend

The backend stores each simulation in a RESTful API, which then links to the name of the simulation and each of the names and scores of all the high scores. The website backend then passes

this information on to the angular frontend, which displays these scores on the high score table next to the game window.

When a Unity simulation determines it is appropriate to save a high score into the server, it begins by constructing a GET request to the backend Spring API endpoint responsible for storing a new datapoint, using a URL. It then sends the data to Spring by constructing a Unity Web Request from the URL and performing the request, which results in the server receiving the data and storing it into its internal data storage system. By waiting for the web request to finish, Unity can verify that the data was stored correctly, as otherwise an error would be returned by the Unity Web Request. Once the data is sent to the Backend Spring code, control of the information passes off to Spring. This is displayed in figure 3, where the first arrow represents when the player dies, a GET request is sent to the server. The next arrow from the top then shows the information stored in the GET request. The information stored from the GET request corresponds to the Simulation name, the data tag within the simulation, the user's name, and the score from the user. The last arrow shows the GET request information sending information to the backend through a method called storeData().

Once Spring receives the data from the WebGL game, it then serializes the data to disk. Even though the exact details of how the data is stored to disk is not the focus of this project, it is worth mentioning. Since any database code would likely need to be rewritten by PS Technology to work with their specific systems, we opted to create a class that has two function calls that can be edited to save and load data, using whatever system the client desires. By default, it simply saves the data to a file on disk in the same folder as the jar. Spring then holds onto this stored data until the frontend requests the data, at which point it loads the data and returns the requested subset of the data to the website for processing.

To get every recorded score for a particular simulation Angular uses a special service to make a GET request to the Spring API. The API returns a JSON containing, among other items, a list of key-value pairs corresponding to individual players and their highest score. When Angular gets this document it strips it of everything but the scores. An array consisting of a dedicated class for representing these scores is populated from the shortened JSON and then sorted in descending order numerically. This is all done during the ngOnInit() of the Simulation component so that the webpage will be able to display this information to the user once it loads.

Dynamic Assets

PS Technology asked that we implement a method to allow for the dynamic loading of assets, or the components that make up their Unity simulations. We were informed that PS Technology creates and loads their assets from asset bundles in their simulations. Asset bundles package groups of assets together. Since they would normally load their assets at runtime of simulations from these bundles, to reduce the amount of code PS Technology would have to change, we designed a similar method to load asset bundles from a web server.

First, we had to figure out whether or not asset bundles created for the typical Unity Windows application would be loadable from a WebGL format. After some experimentation, we learned that the asset bundles need to be recreated specifically for a WebGL format with specific compression setting changes. We assumed that PS Technology's simulations would load their asset bundles from relative file locations, so we first attempted to load these WebGL formatted asset bundles from relative file locations on the server.

Loading the asset bundles from relative file locations would require the least amount of code changes for PS Technology, so this first approach was the most logical. This method also allowed us to manage the file locations of asset bundles that were in either a simulation's specific asset bundle folder or the common asset bundles folder. After experimenting with these asset bundle loading methods, we found that the WebGL formatted asset bundles could not properly load off of a relative file location on a web server. Our next approach was to see if we could load the asset bundles from specific web addresses rather than relative file locations.

While loading asset bundles from specific web addresses would require PS Technology to create a method or obtaining these web addresses, it was the most efficient method of loading simulation asset bundles at runtime. This way, as long as the individual simulations were provided the proper web addresses, they could access the bundles through the web server. Unfortunately, we did not find a way to simultaneously manage the file locations of the asset bundles, but this was a small sacrifice that puts slightly more responsibility on PS Technology for managing the locations of the asset bundles they would like to be used in their simulations.

As shown in figure 4 below, altering the URL location of the asset bundles that a simulation is loading can change which asset is being loaded at runtime of a simulation. In figure 4, the "CommonAssets" folder location contains the pick-up material (pickupmat) bundle with a green color compared to the "Simulation1Bundles" folder location that contains a pick up material with a yellow color.

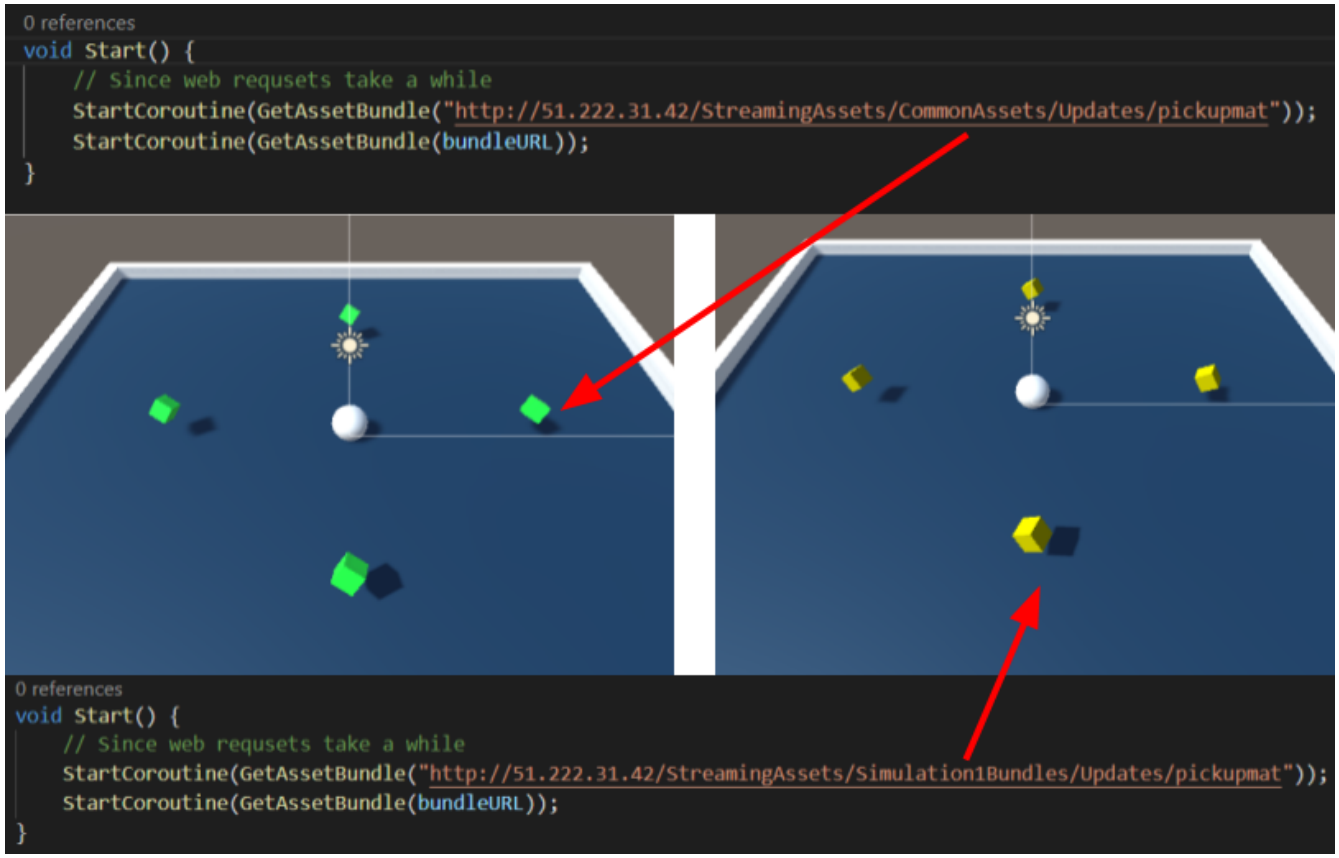

Figure 4: Dynamic Asset Bundle Loading (Common Assets versus Specific Simulation Bundles)

Quality Assurance

The first element of our Q&A plan was the manual testing of the website. As the website framework was developed, users navigated and tested the pages to see if the simulations were properly running, there were proper interactions between the simulation receiving and sending out data, and that the website was intuitive to navigate. This helped to ensure that the UI and UX of the website were clear and easy to use, and to make sure users wouldn't get bogged down with unnecessary details.

Since large elements of our project involved integrating separate components, we invested a fair amount of effort into creating and carrying out our integration plan. Primarily, the code is regularly deployed to a testing server where all the elements are integrated together to ensure all components properly communicate. This helps to ensure that each of the components works together as they developed, to prevent the elements from making invalid assumptions about other components. It also helps to ensure that all the code properly compiles and functions as intended in the real world.

To help ensure that all the code is of sufficiently high quality, can be easily understood, and functions correctly, all code must undergo code review before being merged into the master branch. This was accomplished by requiring all code to be pushed to a new branch and then requiring the pull request to be reviewed and approved by a team member who didn't write any of the code before it was merged into the master branch. This ensured that all code committed to the master branch was of sufficiently high quality.

It was very important during the development of this project to ensure that all the solutions found for the technical problems were acceptable to PS Technology. As such, during development, the product was regularly demonstrated to the client to ensure it would meet their expectations and follow their vision for the project. This also helped to keep the project on track, following the client requirements, and helped to prevent scope creep from unneeded features.

Since maintaining the security of the website was important for the project, being a live website, we employed two forms of static code analysis. This is primarily the fact that the git repository being regularly scanned for libraries and dependencies with security holes, as well as checking for common coding errors that can introduce security holes into the project. This helps to ensure that the product we ship meets a high level of quality for the security of data on the website and the web server and that the website doesn't put users or their information at risk.

Results

Due to time constraints on this given project, we had to choose to cut out some content for the final release. One of these included the addition of a moderator or admin to create buttons and bind actions to these buttons for each individual game page directly through the website UI instead of manually programming buttons through the HTML files. Another feature we had to stop on development was creating a system to display a thumbnail for a given simulation on the dashboard page. These thumbnails would provide more context to the user about each simulation, but as it stands, we can hope to integrate thumbnails in the future. Furthermore, transitioning to the Azure web servers would have been possible with more time, but due to the minimal time on the project, we decided it would be best to stay on the OVH servers.

All performance testing was based on small-scale stress tests on the OVH server. Since the web server that was used to host our framework was relatively low powered, we had to ensure communications between multiple users and the website were stable. This was tested with multiple users sending requests to the website at once, which the server handled with relative ease.

Since our project did not use any form of automated testing, all tests were done manually. One of these tests included modifying a specific simulation built for dynamic loading that will cause the next instance of the Unity simulation to load the proper assets according to the changes on the web server files. A second test was observing if our front end was properly loaded in from the web server and displayed properly on the main server page. Both of these tests were performed by acting similarly to a user of the website, and the results were that the code and systems functioned properly.

Other forms of testing included our results of usability tests which were also tested manually. Some of the results of our usability tests included that we manually navigated through our web pages to ensure that all of our router links worked correctly, that the layout of the website itself was not complicated, and that the website provided a user ease of accessibility. Following this testing, we concluded that the layout and format of the website were acceptable. We created a simple UI using a navigation bar so the user could access any page with no problems. Each of these systems was also created with the intent to give our clients a base to use and/or extend them in the future.

To expand on our project in the future, we can implement additional features to our website such as reformatting to different devices or screen sizes. These will be updated according to the user's resolution. As of now, we currently have one simulation being loaded into a webpage, but in the future we can include the embedding of multiple simulations into a single webpage and having the two simulations interact with each other as well as interacting with the website. Additionally, we can improve the server of the website in the future. As user traffic grows, we can conduct large-scale

stress testing on the website to ensure the servers can handle a large number of users at the same time.

Over the course of the field session, we have learned several lessons while developing this project. One of these lessons included that scope creep can cause elevated stress in a project. Even though we had a project timeline planned out for our tasks, we did not guarantee that they would be finished at the exact deadline given to them. We also learned that we could not assume how long a task would take, instead we could only give a rough estimate and plan accordingly if things pan out to take much longer than expected. Lastly, after going through the learning process of new technologies and beginning to designate responsibilities, we were open to exchanging tasks with group members as we became more comfortable with using different software and technology.

Appendices

Documentation

The full documentation for this project is linked below. This was not embedded in the report as it is a nearly 30-page long document. It is an HTML document that was compiled from markdown using the marked command-line tool.

Included in the documentation are deployment instructions for the codebase, a project summary, as well as descriptions of how each component is implemented and how each component is interlinked. The documentation webpage contains internal links to aid with the navigation of the document.

https://14erc.com/PSTechnologyWebGLDashboardDocumentation/

Website Layout

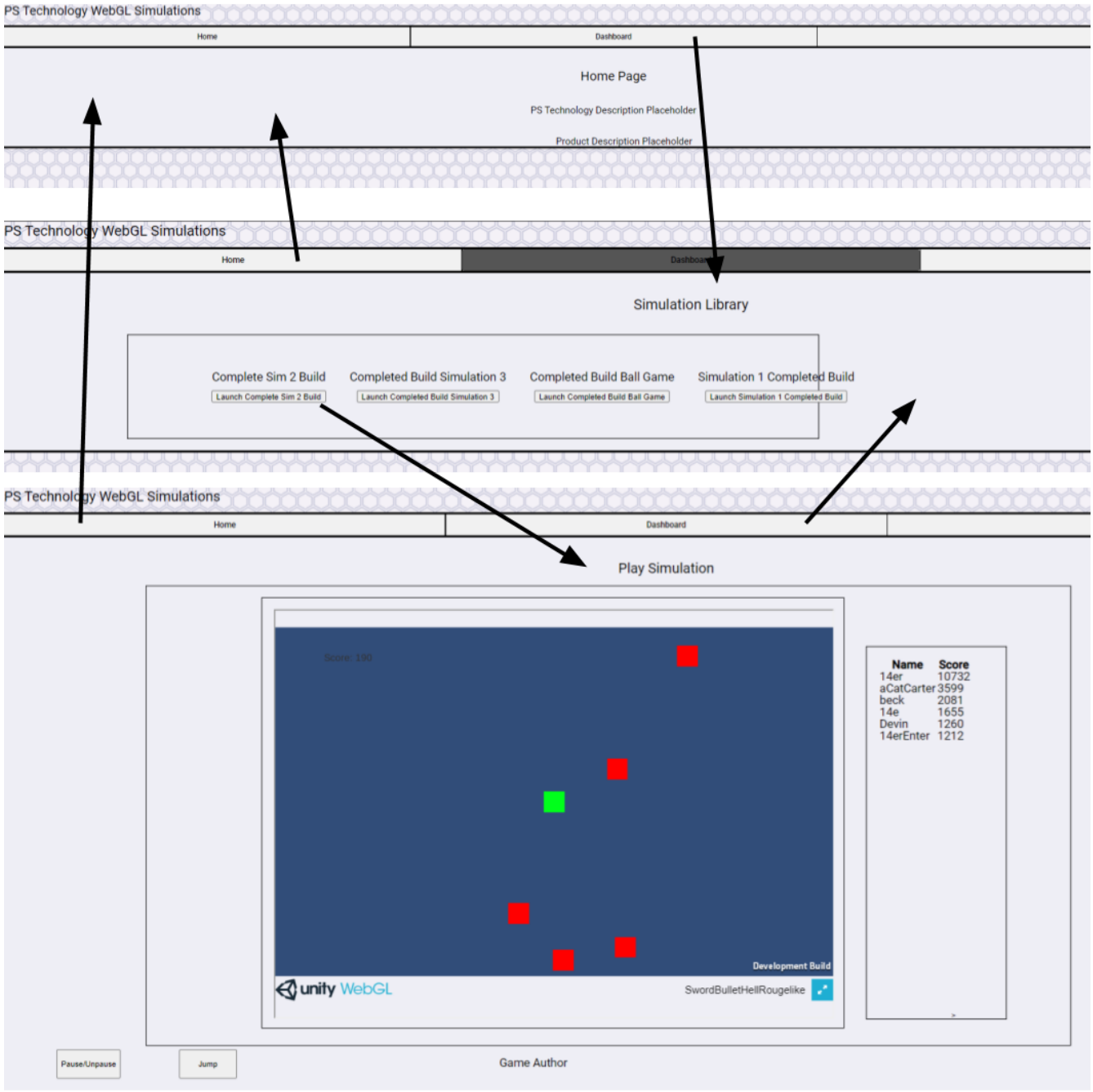

Figure 5: Website Layout

Each arrow represents where clicking on the specified button will take you within the application.