CS Senior Capstone Fall Term - Project Report

December 2nd, 2017

CDK Global: No more touch. No more Keyboard. Bring it All Together. Using Technology to Teach Humans.

Prepared for

CDK Global

Trevor Moore ______________________ Signature ________ Date

Prepared by

Group 9

Look Boss, No Hands

Brandon Dring ______________________ Signature ________ Date

Nipun Bathini ______________________ Signature ________ Date

Carl Benson ______________________ Signature ________ Date

Abstract

This document seeks to explain what exactly we have been doing for the past 10 weeks with regards to the Look Boss, No Hands project. We will go over exactly what the project is trying to accomplish, and the weekly status updates on progress that lead up to where we are now. Then, a short excerpt talking about where the project is in its current form. Afterwards we will explain any problems we have encountered so far, and any interesting findings so far. Lastly, we will go over a retrospective on what we've done as a team this semester, and ways we can improve.

Contents

1 Introduction

2 Alexa - Nipun Bathini

2.1 Current Status

Last term, we collected our Alexas from the client and began implementing our own code into a skill. A majority of fall term involved the team familiarizing ourselves with the Echo, and creating basic applications that were unrelated to our project. We later began research we needed to prepare ourselves to implement an Alexa skill for our own project, including finding Artificial Intelligence and Machine learning packages for Alexa. From this we learned that Alexa supports setting and passing along session attributes in the form of key: value pairs. The sessionAttributes are retained and passed along through the session. The session continues until the buildResponse receives a true value for shouldEndSession. The functions that will take the requests can check these values and if they are current they can be used to fill in any blanks in a future request. This was a crucial find, as it will be useful all the way until our stretch goal, the wearable. The wearable will need the machine learning to be able to identify important trends to alert the user. Next, the main purpose of the entire project is to visualize data which is being pulled from a MongoDB database. In our project, Alexa acts like a middle man, between the database and the virtual reality headset. The database was populated with fake data, will contain real data once the client takes control, containing various sales information about dealerships and their vehicles. The return of mongoDB data from the Alexa request was implemented, so when the user makes a voice request to get sales data for a certain maker and time period from the database, Alexa is able to return it. Returning to artificial intelligence, Alexa was programmed to track popular requests which will be stored in the database for future AI implementation. We did not want Alexa to have basic responses, making it sound like a robot, and focused on making the device to be able to talk more or less like a human. This was done by providing Alexa with a semi-intelligent response guide to follow such that it provides logical responses to any question asked and the same with error handling. We later further improved the responses so when the user is querying data from Alexa, it should respond with the highest and lowest numbers of the query to speak back to the user. Originally if anyone is on the web socket port they will all receive the same message whenever anyone makes a request via the Alexa. A user should not be bothered by others using the Alexa, so we set up socket.IO rooms such that there is a private message line that allows multiple people to be using the project at once. In order to make sure Alexa was doing everything we expected it to, we implemented tests, including how Alexa was interpreting commands. Instead of allowing Alexa to take in random requests we defined a data model, and actually planned out what Alexa commands should be used and the capabilities of the system. Overall, Alexa has been working how we have been expecting it to, and can be seen in our demo.

2.2 Left to Do

Although, we have done a lot with Alexa already we have a few more implementations that we would like to complete. We plan to have Alexa pull the last user state from the database and append it to the current request. Before each request, using the userID we will pull the last request that they had made. Then use the data that isn't null to populate the same fields in the current Alexa request before getting new data. Currently, if you're looking at a city/state combo, Alexa will try to refine your query further. We will reset the filters to cancel out refinable routing, making it so there is an option to go back looking at just the state data. Alexa should also be able to give the user suggestions by using the highest and lowest numbers found for similar queries, after the current data has been said to the user. Alexa will then ask them if they would like to see similar data regarding the high/low numbers of other cities and states. We have already implemented some tests for Alexa, but theres always more tests we can write! Fortunately, AWS provides a testing tab, which we plan to use to create all tests possible for Alexa and ensure that the proper responses are received. In our project we implement Wrld Maps, which is a 3D maps visualization platform that we use in our VR world. Now that we have added graphs to visualize the data that we can map to in VR via Alexa commands, we need a specific wake command to have VR switch cameras to the Wrld maps. The command should also do the aggregations for the pie, and bar chart near the location on the Wrld map

2.3 Problems

One of the first problems we ran into when working with Alexa, was the data that it would be retrieving from the database. Our client is not going to provide us with their real sales data, since it is confidential, but can be implemented once CDK takes our project. Since claiming an international companies sales as different as their true value is illegal, we were not allowed to have a database populated with real manufacturers and models. In order to work around this, we filled the database with fake make and models. When the database was first filled, every city only had one dealership. The data we had was sufficient to show off our project, but in reality there are going to be more than one dealerships in a city. We added more dealerships to cities, and even add latitude and longitude, so that way we can move the camera in the mapping environment to the exact position of the dealership. In addition, before each response when replying to a request by the Alexa, we will add a reminder saying 'please remember this is fake data' into the Alexa response. We also had issues with multiple users on the same websocket port, and to avoid these problems we created Socket.IO rooms to allow multiple users at once

2.4 Interesting Code

```
/**
* Called when the user specifies an intent for this skill.
*/
onIntent(intentRequest, userID, callback) {
    const intent = intentRequest.intent;
    const intentName = intentRequest.intent.name;

    // Dispatch to your skill's intent handlers
    if (intentName === 'Capstone')
        this.routes.goToRoute(intent, userID, callback);
    else if (intentName === 'AMAZON.HelpIntent')
        startStop.getWelcomeResponse(callback);
    else if (intentName === 'AMAZON.StopIntent' || intentName === 'AMAZON.CancelIntent')
        startStop.handleSessionEndRequest(callback);
    else
        throw new Error('Invalid intent');
}
```

3 AWS - Carl Benson

3.1 Current Status

Two primary systems for the project are powered by Amazon Web Services (AWS): Elastic Compute Cloud (EC2) and MongoDB. EC2 hosts the server that acts as an intermediary between the Echo, database, and Vive. After requests are processed by the Echo system, they are passed on to the EC2 server through query parameters, with content such as requested state, city, and time period. With the request, a unique identifier is also passed along. This identifier is used later on for logging and organization functions. The query parameters are aggregated into a single query that is passed to the database. The database returns any data that matches the request. When a request is processed, it is also logged. This will allow for future development of suggestions by Alexa. The logging takes the identifier, request type and request parameters. The EC2 portion is written using Node JS.

The database is using a MongoDB instance hosted on AWS. Using MongoDB provides a straightforward method for storing and retrieving data. The Echo sends the request formatted as a JSON, with values in the form of state: OR, which is the same format that MongoDB queries utilize. In order for data to be retrieved from the database, data must first be stored in it. In order to create a large enough sale database, a Python script was created to enable the creation of large amounts of pseudorandom information. The script selects a random amount of cities to create for each state, and then for each city creates a random amount of dealerships. Once the dealerships are created, they all receive randomly generated sales, including details such as date, brand, sale price, owner, and color. The sales are stored in an array within the corresponding dealership object. Storing sales this way allows for more flexibility when requesting data. Dealerships as a whole can be retrieved along with their sales, or sales from multiple dealerships can be queried for attributes.

Communication between the EC2 server and the computer running the HTC Vive is performed using the SocketIO library. The library utilizes websockets to create and maintain a connection between two systems as well as handle data transmission. Since the returned data from the database is already formatted as a JSON, it can be passed directly on via SocketIO.

3.2 Left to Do

Communication between the Echo and EC2 system has all the basic functionality in place. The largest possible future improvement is to increase or improve the query system. Currently, parameters are passed along using the form of /city/state?city=portland&state=oregon. This can be improved through the inclusion of further path options, such as time, brand, or price, or by switching to an adaptive system that can determine the proper database query from whatever parameters are passed to it. So rather than requiring predefined routes, a single route can take variable parameters.

Unless any new features require further details, the database and generator script are finished. The database is populated with thousands of sales created by the generator, each including the information required to be represented in the VR environment.

Through the use of logging, suggestions can be made to the user. These suggestions could be possible searches, direction towards information, or explanations of presented data. For example, if a user requested a chart of sales by brand in Corvallis, OR, Alexa could provide an overview of what is being shown as well as how the data is represented on the graph. Alternatively, if in a previous session the user made a particular request, Alexa could suggest the same request again to allow for easier access on subsequent sessions.

Communication between the EC2 server and Vive client currently only supports a single client at a time. This is due to the SocketIO communication broadcasting messages to all connected clients rather than allowing direct messages.

Previously, the unique identifier passed along by the Echo was intended to be used to create individual SocketIO rooms. When connecting, the Vive client would send a matching identifier, which the server would respond to with the appropriate room to join. This would allow the server to match room numbers to identifiers and send data only to the matching client. However, the implementation for Unity does not include the functionality for rooms.

3.3 Problems

The current problem is the lack of rooms in the SocketIO implementation for Unity. One way, albeit an inefficient one, would be to include the identifier in each data transmission. This would allow each client to ignore any data not destined for it. However, the down side to this is data usage could rapidly increase. While for a couple of low activity clients the system wouldnt be much less efficient than rooms, if a large amount of high activity clients were to all connect simultaneously the amount of data could become a concern. However, for the scope of this project it is unlikely to become an issue.

3.4 Interesting Code

Part of the VR experience is a flyover style map that can take you to a dealership on request. In order to do this, the dealership objects stored in the database need to include accurate, and unique, coordinates to let the mapping library know where to go. While each city has latitude and longitude of city center included, it wouldnt be useful, practical, or appealing to have dozens of dealerships layered on top of each other on a single point. This creates the need to offset the location dealerships are created at from the location of their city.

```
r_earth = 40074

#Calculate the circumference of earth at current location
#needed for finding lat/long offset from miles
cir_earth = r_earth * math.cos(float(locations[loc]["lat"]))

#KM (used in lat/long) to miles
conversion_factor = 0.62137119

shift_lat = float(np.random.normal(0, 5.0, 1) / conversion_factor)
shift_lng = float(np.random.normal(0, 5.0, 1) / conversion_factor)

lat = (((float(locations[loc]["lat"]) + shift_lat) / 40075) * 360)
lng = (((float(locations[loc]["long"]) + shift_lng) / 40075) * 360)

dealerships.append({
    'lat': dealer_lat,
    'lng': dealer_lng
})
```

The code picks two random numbers using the random.normal function. This random normal distribution returns numbers that are clustered towards the center of the range. For any given city, approximately 68% of dealerships will be within 5 miles of city center, 95% will be within 10 miles, and 99.7% will be within 15 miles. This creates a city with the majority of dealerships near city center, but will still include ones further outside city limits. This helps to improve the realism of the random dealership generation.

4 VR - Brandon Dring

4.1 Current Status

We have finally moved out of the proof of concept phase, away from simply creating basic objects that are grabbable like squares and spheres. The P.O.C that we created was great for learning the basics on how to develop in Unity with C#. So far through this semester we have swapped out the basic shapes on a flat plane to instead include a virtual house. This virtual house looks like a stereotypical fancy house one might find to help contribute to the unique environment of viewing the data. Inside this house there are a myriad of graphs of different types found from the Chart and Graph library in the Unity asset store.

Ive also implemented with each chart its own socket listener that waits for an event from our EC2 server. Upon receiving an event on the listener it retrieves the data that has been pushed through the socket and updates the graph accordingly.

There is also some headway being made, or atleast started on regarding the use of the WRLD 3D mapping library, we have the ability to switch to the WRLD map by simply having the VR headset listening to yet another socket listener. Upon hearing a socket event, the headset then switches from the VR camera to a camera that is controlled by the WRLD map to change environments.

4.2 Left to Do

The Vive seemingly is the single area with the most left to do, as the project is more or less centered around the VR experience. To start with we need to enable the user to switch between the home view that we currently have established. Where they have a myriad of graphs that they can view at once such as: bar graphs, line graphs, pie graphs, etc. To a street view, where theoretically if the database was populated with real sales data, a user could see their dealership in VR. Accompanied with the viewing of their dealership, they should also be able to see basic stats in the classical graphing examples of just a bar and pie graph. This is more of a neato feature, that the client has expressed interest in. With that, we will need to figure out how to implement a virtual mapping environment, along with adding objects to said virtual map.

There was also some breakage that had happened during development. The basics of VR, such as locomotion or even being able to grab things having broke for some reason or another when adding the latest features. So obviously, we need to reimplement the VR functionality or its just going to be a TV screen too close to your face.

4.3 Problems

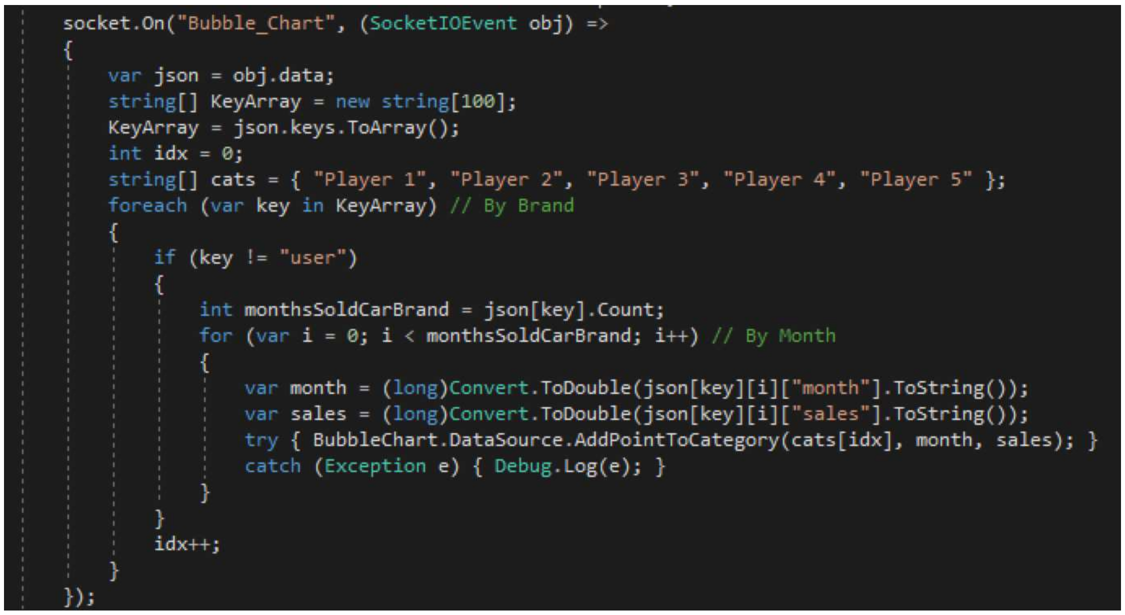
Using Unity to develop a VR experience has proven fine for the most part, the C# API that they use is more or less simple to use. With the caveat of parsing through JSON in C#, thanks to types. Its hard to describe and plan the structure of a variable length JSON object, being that it can contain nested arrays of objects full of strings, booleans, and numbers.

However, when trying to learn C# at the same time as trying to learn a 3D modeling environment such as Unity, it proves rather difficult. You throw some not perfectly documented Unity libraries on top of that, it proves to be at the very least time consuming.

Also, trying to resolve merge conflicts in Unity is next to nearly impossible, I am for one baffled at how people use Unity in a enterprise setting with the rudimentary version control that they have. Every time that we have had a merge conflict it has been a 50,000 line diff of a file that we seemingly havent even touched, its just a metadata file that is used by Unity itself. So instead every time that we do have a merge conflict we just revert to the last non-conflicting commit in a very barbaric and ancient way.

The two main libraries that we use for the project being Graph and Chart in the Unity asset store, and the Wrld 3D maps also in the asset store. Both of the libraries have proven to contain horrid documentation on basic use, or extensibility of what you can do. The prior is due to the graphing library being done by a John Doe as a side project, who probably created the project without the intent of it being used for our purposes. The latter being a port from a HTML based mapping library by the same company. Clearly though the Unity port was an after thought of the library, as the functionality demoed on the website, along with the good part of their documentation is based on the javascript library. What this has led to is actually having to dig through the source code to try and understand what a specific function can do. Even having to using the Visual Studio autocomplete to browse the library the easiest, instead of simply just reading the docs.

4.4 Interesting Code



```
socket.On("Bubble_Chart", (SocketIOEvent obj) =>
{
    var json = obj.data;
    string[] KeyArray = new string[100];
    KeyArray = json.keys.ToArray();
    int idx = 0;
    string[] cats = { "Player 1", "Player 2", "Player 3", "Player 4", "Player 5" };
    foreach (var key in KeyArray) // By Brand
    {
        if (key != "user")
        {
            int monthsSoldCarBrand = json[key].Count;
            for (var i = 0; i < monthsSoldCarBrand; i++) // By Month
            {
                var month = (long)Convert.ToDouble(json[key][i]["month"].ToString());
                var sales = (long)Convert.ToDouble(json[key][i]["sales"].ToString());
                try { BubbleChart.DataSource.AddPointToCategory(cats[idx], month, sales); }
                catch (Exception e) { Debug.Log(e); }
            }
        }
        idx++;
    }
});
```

This is how we populate charts in the VR environment. First, we have to declare what our socket listener is as we did above calling it '$Bubble_Chart'$. Then we have a callback function that handles the event. The main difference between all the graphs here is just trying to loop through the JSON array has proven very difficult. To get around how types are implemented in C# and trying to access them through a various JSON structure I merely just looped through the elements in an array. Then after accessing the data that I need, I convert the type from string to long, and add the data to the graphing library to be interpreted. Something that should be really simple, was rather difficult coming from an untyped and interpreted Typescript { Javascript background.

With JSON data types that could have a variety of different structures, it has forced me to keep the data structure as simple as possible. Which is in part a good thing, but it also prevents from creating more complex data structures. Along with preventing adding the use of metadata to the objects passed through the sockets.

5 Conclusion

Given our current stage of development, we feel confident that we will meet our requirements before the final deadline. We have the core functionality in place. From here, we will work on bug solving, efficiency, and additional functionality to provide a better experience and end result. Our most recent demo with our client, where we showed them our current progress left them thrilled.